UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Dominik Sedmak

Arhitektura sistema v prevajalskem podjetju

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2016

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Dominik Sedmak

Arhitektura sistema v prevajalskem podjetju

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: dr. Andrej Brodnik

Ljubljana, 2016

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Sistemska arhitektura v splošnem sestoji iz zunanjega območja in notranjega območja ter vmesnega demilitiziranega območja. Seveda se omenjena sistemska arhitektura mora prilagoditi konkretni dejavnosti podjetja. V diplomski nalogi obravnavajte podjetje, ki se ukvarja s prevajalsko dejavnostjo, kar pomeni, da so njihov glavni vir prevajalci. Posledično mora arhitektura omogočati preprost dostop prevajalcem do nalog na eni strani in po drugi podjetju preprosto upravljanje z viri. Pri pregledu in predlogu se osredotočite predvsem na kakovostne kazalnike, kot so odzivnost, povečljivost, prilagodljivost (skladnost), povezljivost ter predvsem varnost.

Zahvaljujem se podjetju Iolar, brez katerega tega diplomskega dela ne bi bilo, mentorju dr. Andreju Brodniku za pomoč pri sestavljanju dela in vso ustvarjalno kritiko. Poleg tega pa bi se zahvalil kolegom Juretu Roglju, Mateju Gašperšiču in Samu Kralju, ki so projekt začeli, in pa seveda svojim staršem za vso podporo.

Podjetju Iolar.

Kazalo

Povzetek

Abstract

Poglavje 1 Uvod 1

1.1 Struktura naloge 2

Poglavje 2 Orodja in tehnologije 3

2.1 Razvijalska orodja 3

2.2 Delo s podatki 5

2.3 Ogrodja 6

2.4 Podatkovni formati 7

2.5 Standardi za overjanje in pooblaščanje 8

Poglavje 3 Arhitektura sistema 11

3.1 Splošna teorija dobre arhitekture 11

3.2 Obstoječa arhitektura 13

3.3 Načrtovana arhitektura 14

3.4 Jedrni sistem 15

3.4.1 MVC obličje – spletna aplikacija 15

3.4.2 WebAPI zaledje – DBAPI 17

3.4.3 IdentityServer OP-strežnik 19

3.4.4 Strežnik podatkovne baze 20

3.5 Širša arhitektura sistema 22

3.5.1 Podpora projektnemu vodenju 22

3.5.2 Projetex 23

3.6 Overjanje in pooblaščanje 23

3.6.1 Pridobitev žetona 23

3.6.2 Uporaba žetona ... 24
3.7 Prenos podatkov ... 24
Poglavje 4 Kvalitativne lastnosti sistema ... 26
4.1 Odzivnost ... 27
4.1.1 Obstoječa arhitektura – težave ... 27
4.1.2 Načrtovana arhitektura – rešitve ... 28
4.2 Povečljivost ... 32
4.2.1 Obstoječa arhitektura – težave ... 33
4.2.2 Načrtovana arhitektura – rešitve ... 34
4.3 Prilagodljivost ... 36
4.4 Povezljivost ... 37
4.5 Varnost ... 38
4.5.1 Vrinjenje ... 38
4.5.2 Napačno zasnovano overjanje in upravljanje s sejo ... 39
4.5.3 XSS-ranljivost ... 40
4.5.4 Nezavarovane reference na podatke v zaledju ... 41
4.5.5 Napačne varnostne nastavitve ... 42
4.5.6 Manjkajoč nadzor dostopa na nivoju metod ... 43
4.5.7 Ponarejanje spletnih zahtev (CSRF) ... 43
4.5.8 Nepreverjene preusmeritve in posredovanja ... 44
Poglavje 5 Sklepne ugotovitve ... 45

Kazalo slik

Slika 2.1 Vmesnik Visual Studia. 3
Slika 2.2 TortoiseSVN vmesnik. 4
Slika 2.3 IIS Vmesnik. 5
Slika 2.4 Delovanje Entity Framework ogrodja. 6
Slika 2.5 Primer JSON formata. 8
Slika 2.6 Format JWT [2]. 8
Slika 2.7 Potek pooblaščanja s prijavnimi podatki lastnika vira [2]. 9
Slika 3.1 Diagram arhitekture, kot je bila postavljena. 13
Slika 3.2 Diagram arhitekture celotnega sistema. 14
Slika 3.3 Postavitev spletne aplikacije v celotni arhitekturi. 15
Slika 3.4 Prvi korak izdelave profila. 16
Slika 3.5 Postavitev DBAPI-ja v celotni arhitekturi. 17
Slika 3.6 Postavitev overitveno-pooblaščevalnega strežnika v celotni arhitekturi 19
Slika 3.7 Relacijsko entitetni model podatkovne baze. 21
Slika 3.8 Postavitev SISI v celotni arhitekturi 22
Slika 3.9 Projetex v celotni arhitekturi 23
Slika 4.1 Poenostavljen primer združevanja funkcij vmesnika. 29
Slika 4.2 Primer združevanja klicev v podatkovne baze. 30
Slika 4.3 Primer uporabe predpomnjenja. 31
Slika 4.4 Primer uporabe asinhronega klicanja. 34
Slika 4.5 Razlika med overjanjem s piškotki in overjanjem z žetoni [2]. 35

Seznam uporabljenih kratic

kratica	angleško	slovensko
IDE	Integrated development environment	Integrirano razvojno okolje
RDBMS	Relational database management system	sistem za upravljanje z relacijskimi bazami
ORM	Object-relational mapping framework	Objektno-relacijski pretvornik
EF	Entity Framework	Entity Framework
SOA	Service oriented architecture	Storitveno usmerjena arhitektura
SOAP	Simple Object Access Protocol	Preprosti protokol za objekte
REST	Representational State Transfer	Predstavitveni prenos stanja
STS	Security Token Service	Storitev za izdajo varnostnih žetonov
JWT	Json Web Token	Json spletni žeton
DMZ	Demilitarized zone	Demilitarizirano območje
DTO	Data Transfer Object	Objekt za prenos podatkov
API	Application Programming Interface	Vmesnik za delo z aplikacijo
LINQ	Language-Integrated Query	Poizvedba vgrajena jeziku
JSON	JavaScript Object Notation	JavaScript oblikovanje objektov

Povzetek

Naslov: Arhitektura sistema v prevajalskem podjetju

V delu je predstavljena arhitektura sistema, ki je bil zasnovan in izdelan za potrebe podpore delovanja aplikacij v prevajalskem podjetju, prek katerih se izvajata dva ključna procesa v podjetju. Ta procesa sta upravljanje s človeškimi viri in vodenje projektov. Ciljne kvalitativne lastnosti sistema so odzivnost, povečljivost, prilagodljivost in še posebej varnost. V ta namen arhitektura sestoji iz več relativno neodvisnih komponent. Obličje, ki temelji na ogrodju ASP.NET MVC in dela z uporabnikom za potrebe izvajanja procesa upravljanja s človeškimi viri, zaledje, ki temelji na ASP.NET WebAPI tehnologiji in deluje kot programski vmesnik za dostop do podatkovne baze, aplikacija IdentityServer, ki deluje kot storitev za izdajo varnostnih žetonov, in pa sama podatkovna baza Microsoft SQL Server, ki shranjuje podatke. Poleg omenjenih komponent, ki predstavljajo jedro sistema, sta v sistem vključeni še dve aplikaciji: SISI, ki je namizna Windows Forms aplikacija in je namenjena delu z uporabnikom, njen cilj pa je izvajanje procesa vodenja projektov, ter Projetex, lastniški program, namenjen računovodstvu. V primerjavi s prej postavljeno arhitekturo nova arhitektura občutno izboljša delovanje spletne aplikacije z vidika vsakega od prej omenjenih ciljev arhitekture.

Ključne besede: sistem, arhitektura, ASP.NET, varnost, prilagodljivost, povezljivost, povečljivost, odzivnost, žeton, overjanje, pooblaščanje, API, SOA, REST, JSON, pozaporejanje, asinhrono, piškotki, seja, OAuth, OpenID

Abstract

Title: A system architecture in a translation company

In the thesis a system architecture is presented that was designed and implemented to support certain applications in the translation company, which perform two key company processes. The two processes are human resource management and project management. The final qualitative system properties targeted are responsiveness, scalability, adaptability, interoperability and most importantly security. For this purpose, the system architecture consists of several mostly independent components. A front end, based on the ASP.NET MVC framework that works with the final user carry out the process of human resource management, a back end, based on ASP.NET WebAPI in the role of an application programming interface controlling access to the database, an IdentityServer security token service and the database itself, based on Microsoft SQL Server, which saves the data produced. Besides the components mentioned, which represent the core of the system, two more applications are considered a part of the system. SISI, a desktop Windows Forms application that aims to cover the process of project management and Projetex, a proprietary program meant to perform accounting services. In comparison to the previously established architecture, the new one substantially enhances each of the targeted system properties mentioned earlier.

Keywords: system, architecture, ASP.NET, security, adaptability, interoperability, scalability, responsiveness, token, authentication, authorization, API, SOA, JSON, serialization, asynchronous, cookies, session, OAuth, OpenID

Poglavje 1 Uvod

Iolar je podjetje v procesu racionalizacije in avtomatizacije ključnih procesov. V ta namen se je začel razvoj spletne aplikacije, prek katere bi se bolj optimalno izvajal proces upravljanja s človeškimi viri. Še pred začetkom razvoja spletne aplikacije pa je druga razvijalska ekipa že razvijala namizno aplikacijo SISI, ki naj bi izboljšala učinkovitost procesa vodenja projektov.

Upravljanje s človeškimi viri opravljajo upravljavci z viri. Proces se začne, ko nov izvajalec stopi v stik s podjetjem in posreduje svoje informacije. Te se shranjujejo v raznih formatih, večina v preglednici, datoteke pa v strežniku podjetja. Upravljavci z viri imajo pregled nad temi podatki in datotekami in zagotavljajo, da so pravočasni. Naloga upravljavcev z viri je še, da spremljajo potrebo po prevajalcih znotraj podjetja (za potrebe procesa vodenja projektov) in po potrebi izmed kandidatov izluščijo najbolj primerne ter jih testirajo. S testom se pridobi osnovne podatke o kakovosti prevajalca, na podlagi katerih se potem vodje projektov odločajo o njegovi uporabi v projektih. Testu sledi podpis pogodbe o sodelovanju ali pa tudi ne.

Vodenje projektov opravljajo projektni vodje. Ti opravljajo glavno dejavnost podjetja, in sicer organiziranje zunanjih izvajalcev za prevajanje in pozneje pregled prevodov. Proces se začne z novim naročilom. Projektni vodja pregleda podatke o naročilu in zahteve ter se odloči, če in kako bo delo razdelil in komu ga bo dodelil. Nato dokumente naročila pretvori v prevajalske paketke s pomočjo določenih lastniških programov in jih pošlje izbranim prevajalcem. Poleg tega projektni vodja poskrbi, da delo res poteka, tako da ostane v stiku z izvajalci in poskrbi za kakršne koli težave in za to, da je delo narejeno v roku. Po zaključenem prevodu projektni vodja zbere prevode in jih dodeli pregledovalcem, ki preverijo pravilnost. Ko je prevod zaključen in preverjen, se lahko naročilo zaključi. Trenutno se proces izvaja testno na aplikaciji SISI, dejansko pa je tehnološka podpora procesa razdeljena med več različnih tehnologij, ki niso formalno povezane.

Sistem je sestavljen iz več relativno neodvisnih aplikacij, ki sodelujejo pri zagotavljanju osnovnih storitev sistema. Prva taka storitev je omogočanje zunanjim izvajalcem, da se izdela in uredi profil, zaposlenim pa se omogoči iskanje po vseh profilih in pregled nad njimi. Drugi namen sistema je racionalizacija postopka za izvajanje testiranja kandidatov za potrebe ugotavljanja kakovosti njihovega dela in znanja. Za projektne vodje sistem ponuja storitev

samodejne pretvorbe dokumentov v ustrezne paketke in njihovo razpošiljanje, ker je ročen proces precej zamuden. Poleg tega beleži trenutno tekoče projekte.

Spletna aplikacija je v razvoju že nekaj časa, vendar pa je bila večina pozornosti doslej namenjena delovanju in videzu obličja oz. uporabniku vidnega dela spletne aplikacije, zato je bilo v ozadju veliko možnosti za izboljšave. V okviru diplomske naloge je bil naš cilj načrtovati zaledno arhitekturo sistema tako, da ta učinkovito podpira funkcionalnost obličja. V ta namen so bila postavljena merila, ki bodo podrobneje razložena v predzadnjem poglavju, tukaj pa jih lahko naštejemo: povečljivost, prilagodljivost, povezljivost, odzivnost in najpomembneje varnost. Cilj je bil konkretno izboljšati oceno na vsakem od teh meril v primerjavi z obstoječo postavitvijo.

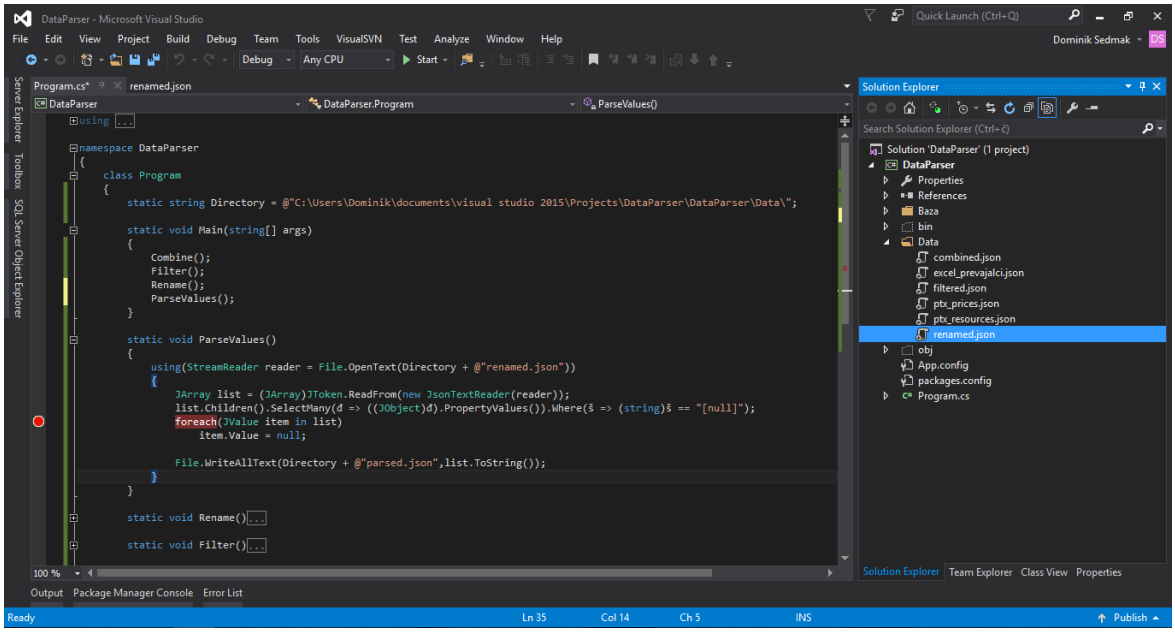
1.1 Struktura naloge

V uvodnem poglavju je na kratko opisano podjetje Iolar d. o. o., zahteve in želje naročnika glede sistema ter cilji oziroma merila, ki bodo pozneje še podrobneje razloženi. V drugem poglavju bodo opisana razvijalska orodja, ki so se uporabljala pri razvoju sistema, in posamezne tehnologije, ki sestavljajo sistem. Nekatere tehnologije bodo v poznejših poglavjih po potrebi še dodatno razložene. V tretjem poglavju bo predstavljen sistem. Opisani bodo glavni gradniki oz. aplikacije, njihov namen in funkcionalnosti, ki jih ponujajo, predstavljena pa bo tudi interakcija med posameznimi aplikacijami v sistemu, protokoli, ki se uporabljajo za komunikacijo, in razlog za to komunikacijo. V četrtem poglavju bo podrobno razdelana arhitektura in njene kvalitativne lastnosti. Ključne odločitve v načrtovanju sistema bodo podrobno opisane glede na to, kako pripomorejo k izvedbi lastnosti sistema. Na koncu bosta v zaključku na kratko orisana pomen sistema za podjetje in delovanje aplikacij, podane pa bodo tudi sklepne ugotovitve.

Poglavje 2 Orodja in tehnologije

Kot je že bilo omenjeno v uvodu, ima podjetje Iolar d. o. o. dolgo zgodovino dela z Microsoftovimi orodji in okolji, poreklo večine programske opreme v uporabi pa je v družbi Microsoft. Podjetje se je odločilo, da bodo za razvoj uporabljena Microsoftova orodja in tehnologije, ker bo tako lahko doseglo najvišjo stopnjo integracije z obstoječimi sistemi.

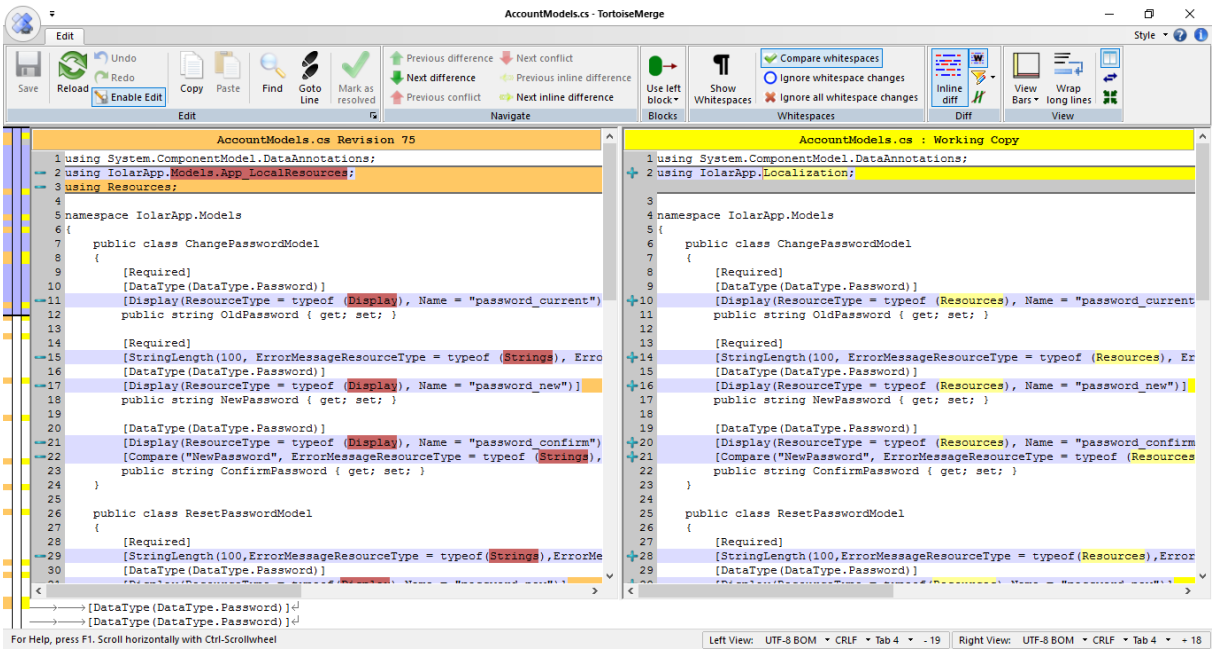
2.1 Razvijalska orodja



Slika 2.1 Vmesnik Visual Studia.

Visual Studio je IDE (Slika 2.1), ki na enem mestu združi več funkcionalnosti, ki jih tipičen razvijalec potrebuje za razvoj kakovostne programske opreme. Vsak IDE lahko deluje le s podprtimi programskimi jeziki in Visual Studio ni nič drugačen. Podpira C, C++, Visual Basic .NET, C# in F#, programiranje pa je potekalo v jeziku C#. Navadno IDE vključuje vsaj urejevalnik besedila z barvanjem kode, vgrajen razhroščevalnik in pa osnovno predvidevanje in označevanje napak. Poleg omenjenih funkcionalnosti Visual Studio ponuja še druge,

predvsem za boljšo povezanost z ostalimi Microsoftovimi izdelki in tehnologijami: npr. delo s povezanimi podatkovnimi bazami in vgrajeno orodje za upravljanje izvorne kode Git. Zelo uporabno je na primer vgrajeno orodje za delo s podatkovnimi bazami in samodejno ustvarjanje kode ter upravljalnik paketov, ki poskrbi, da so zunanje knjižnice pravilno vključene v projekt. Zmožnosti Visual Studia lahko razširimo z različnimi vtičniki. Ker se je za upravljanje izvorne kode uporabljal sistem SVN, je v Visual Studiu prav prišel vtičnik VisualSVN, ki omogoča delo s Subversion znotraj IDE-ja in ReSharper, ki ponuja razne nasvete za izboljšanje kode med pisanjem.

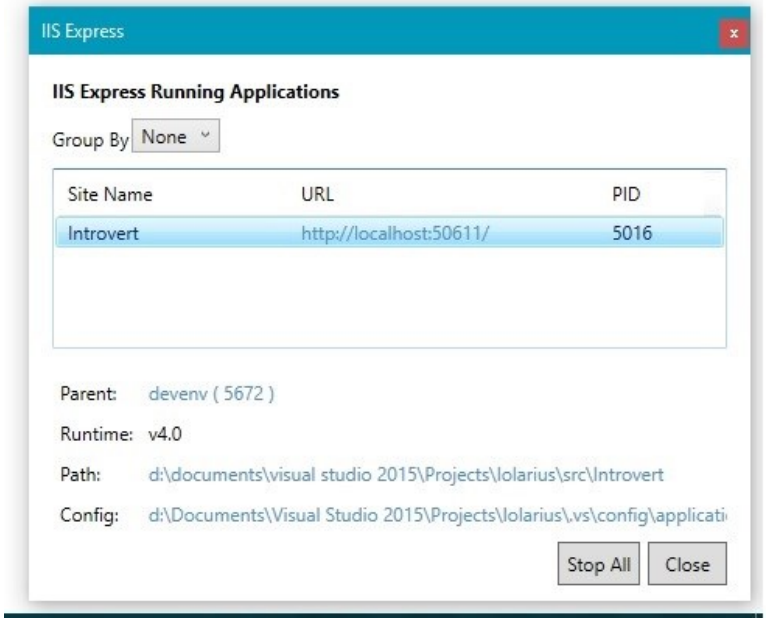


Slika 2.2 TortoiseSVN vmesnik.

SVN (Slika 2.2) je sistem za upravljanje izvorne kode. Omogoča spremljanje vseh sprememb izvorne kode, primerjavo kode pred spremembo in po njej, zavrnitev sprememb na ravni vrstice pa vse do celotnih map projekta. Zaradi teh zmožnosti deluje tudi kot nekakšna varnostna kopija v primeru izbrisa ali kvarnih sprememb. SVN deluje v ukazni vrstici, vendar obstaja več grafičnih vmesnikov, ki naredijo delo s programom precej lažje. Pri delu se je uporabljal TortoiseSVN.

Internet Information Services oziroma **IIS** (Slika 2.3) je Microsoftov spletni strežnik in je konkurenčni strežnik strežniku Apache HTTP. Trenutna različica je IIS 10, ki je vključena v Windows 10 in Windows Server 2016, obstaja pa tudi IIS Express, ki je nameščen z Visual Studiem in prilagojen za potrebe razvoja. Pri delu se je uporabljala lokalna instanca IIS Expressa, za testiranje produkcijskega okolja pa poln IIS. Prednost IIS-ja v Microsoftovem

ekosistemu je integracija z overjanjem Windows in z ogrodjem ASP.NET. Poleg tega ponuja grafično okolje za upravljanje s strežnikom.

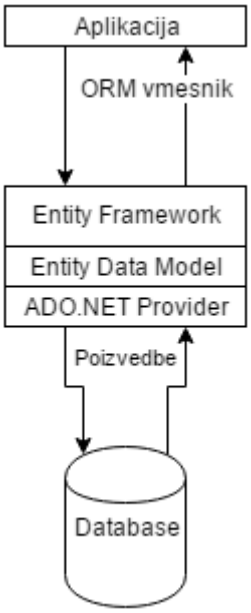


Slika 2.3 IIS Vmesnik.

2.2 Delo s podatki

MSSQL Server je sistem za upravljanje z relacijskimi bazami. Omogoča dostop do podatkovnih baz prek več različnih transportnih protokolov, kot sta TCP/IP ter imenovane cevi in tudi z uporabo deljenega pomnilnika. Skrbi za predpomnjenje podatkov in za spoštovanje principov ACID (atomarnost, konsistenca, izolacija, trpežnost). Pri projektu se uporablja za shrambo vseh podatkov.

Entity Framework (Slika 2.4) je objektno-relacijski pretvornik oz. ORM. Namesto ročnega pisanja SQL poizvedb nam ORM omogoča, da podatke iz baze samodejno prenesemo in pretvorimo v navaden objekt, napolnjen s podatki, oziroma da stanje objekta pravilno prenesemo v podatkovno bazo. EF skrije podrobnosti podatkovne baze in poskrbi za spoštovanje relacij. Ponuja sistem za sestavljanje kompleksnih poizvedb v podatkovno bazo, ne da bi se dotaknili jezika SQL. Uporablja se kot most med podatkovno bazo in programi, ki delajo z njo (WCF, WebAPI).



Slika 2.4 Delovanje Entity Framework ogrodja.

2.3 Ogrodja

Ker je bil sistem zastavljen kot spletna aplikacija, se je za razvoj uporabljalo ogrodje **ASP.NET** [1]. ASP.NET MVC je Microsoftova implementacija MVC oz. model-pogled-krmilnik (ang. *model-view-controler*) arhitekturnega vzorca, ki razvijalcu ponuja različne napredne funkcionalnosti, ki bi jih sicer morali programirati sami. Aplikacija, razvita po MVC-metodologiji, je razdeljena na tri sloje. Model, ki predstavlja podatke in podatkovne strukture, krmilnik, ki jemlje podatke modela ter jih predela in pošlje v pogled ali pa obratno ter pogled, ki podatke predstavi na uporabniku prijazen način. Vključuje tudi tehnologijo Razor pogledov. Ti omogočajo razvijalcu poenostavljeno ustvarjanje pogledov na strani strežnika, kar izboljša kakovost končne rešitve, saj so napake vidne, še preden se aplikacija zažene.

ASP.NET WebAPI temelji na MVC vzorcu in uporablja iste tehnologije kot zgoraj opisani ASP.NET MVC. Njegov namen ni ustvarjanje in prikaz spletnih strani, ampak objava spletnega vmesnika oz. API-ja, ki je dostopen z uporabo običajnih HTTP zahtev in standardnih spletnih tehnologij, kot je JSON in JavaScript. Zato ne uporablja pogledov, tako kot ASP.NET MVC, ampak sprejema in pošilja kar podatke v formatu JSON. ASP.NET MVC se je uporabljal za obličje, torej uporabnik komunicira neposredno s spletno aplikacijo, zgrajeno na tej tehnologiji. ASP.NET WebAPI pa se je uporabljal za zaledje in stoji pred podatkovno bazo.

WCF je ogrodje za izdelavo storitveno usmerjenih aplikacij. WCF omogoča pošiljanje sporočil med združljivimi aplikacijami in gostuje v IIS ali teče kot Windows storitev. WCF je bila

zasnovana za poslovna okolja in je prilagodljiva, da ustreže različnim zahtevam v teh okoljih. Podatki so lahko v XML ali JSON formatu in format samih sporočil je lahko SOAP ali REST. Transportni protokol je lahko HTTP, TCP, imenovane cevi ali kak drug. WCF je bila uporabljena pred začetkom izdelave nove arhitekture.

IdentityServer je odprtokodno ogrodje, ki deluje kot storitev za izdajo varnostnih žetonov. Zgrajena je na podlagi ogrodja ASP.NET v jeziku C#. IdentityServer ogrodje se lahko uporabi za izdelavo storitve STS, ker implementira za to namenjena internetna standarda OAuth 2.0 za pooblaščanje in OpenID Connect za overjanje. Implementira tudi njune razširjene specifikacije. Podpira veliko različnih tipov odjemalcev od navadnih spletnih in namiznih aplikacij do JavaScript odjemalcev, kar pomeni, da je ogrodje zelo prilagodljivo.

JSON.NET je knjižnica za pozaporedenje in razzaporedenje navadnih .NET objektov v format JSON. Odlikuje jo visoka hitrost in podpora .NET-tehnologijam, kot je LINQ. Vsebuje enega od redkih pozaporejevalnikov, ki podpira pozaporedenje podatkovnih struktur s cikličnimi povezavami.

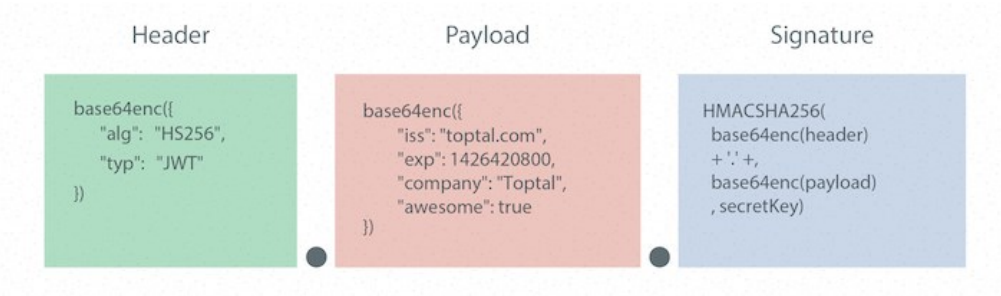
2.4 Podatkovni formati

JSON (Slika 2.5) je format za izmenjavo podatkov. Njegova oblika je vzeta iz zapisa objektov, kot se pojavi v JavaScript, in je zato zelo poznana. Kot format za izmenjavo podatkov se zelo pogosto uporablja in pri enaki količini podatkov skoraj vedno proizvede manjši zapis kot XML, zato se hitro širi v tej vlogi. JSON pozna samo dve podatkovni strukturi – slovar in seznam –, kar prispeva k enostavnosti in univerzalnosti formata. JSON se pri projektu uporablja kot glavni format za izmenjavo podatkov.

```
{
  "array": [
    1,
    2,
    3
  ],
  "boolean": true,
  "null": null,
  "number": 123,
  "object": {
    "a": "",
    "c": "d",
    "e": "f"
  },
  "string": "Hello World"
}
```


Slika 2.5 Primer JSON formata.

JSON Web Token (Slika 2.6) je standardiziran, varen in lahek format za prenos »trditev« (ang. claims). Uporablja se za overitev identitete v okolju OAuth 2.0 in kot varna shramba za informacije o overjenem uporabniku. Podatki v žetonu so podpisani s skupno skrivnostjo ali pa z zasebnim ključem X. 509 certifikata, zato lahko vedno potrdimo, da so podatki nespremenjeni in iz znanega vira. JWT-ji so alternativa bolj tradicionalnim SAML žetonom, ki uporabljajo XML kot temeljni format. Iz tega izhaja veliko večja velikost žetona in počasnejša razzaporeditev vsebine.



Slika 2.6 Format JWT [2].

2.5 Standardi za overjanje in pooblaščanje

OAuth 2.0 [3] je protokol in internetni standard, ki opisuje načine, prek katerih lahko uporabnik (ang. *resource owner*) dovoli neki storitvi oziroma odjemalcu (ang. *client*) dostop do njegovih podatkov oziroma bolj splošno do česar koli, kar mu pripada (ang. *resource*). OAuth 2.0 deluje prek HTTP protokola in je vezan nanj. Po standardu so definirani štirje načini za pooblastitev dostopa do podatkov (ang. *authorization grant type*):

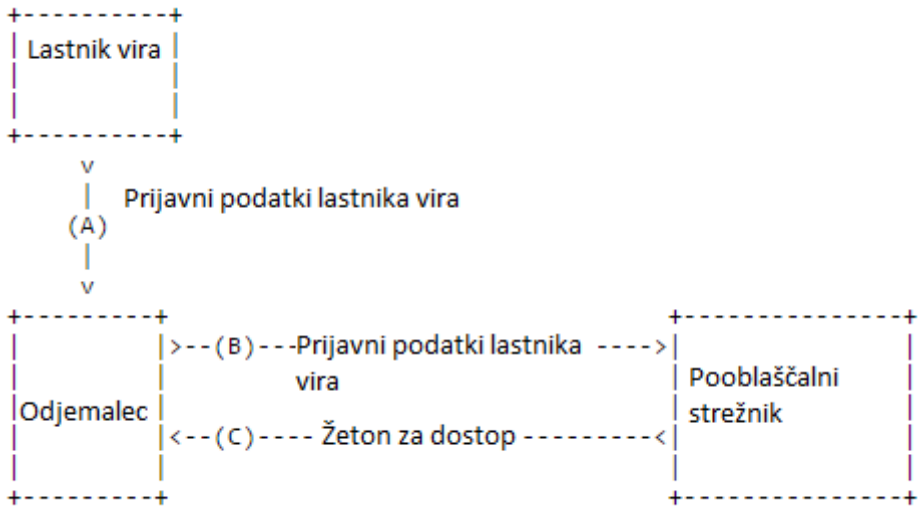
1. Z uporabo pooblastitvene kode (ang. *Authorization Code*):

   Odjemalec uporabnika preusmeri na stran pooblastitvenega strežnika. Ta preveri in pooblasti uporabnika in ga preusmeri nazaj na stran odjemalca skupaj s pridobljeno pooblastitveno kodo. Odjemalec to kodo uporabi, da od pooblastitvenega strežnika pridobi žeton za dostop do podatkov. Na tak način odjemalec nikoli ne dobi prijavnih podatkov uporabnika, kar je dobro, če odjemalcu ne zaupamo. Omogoča tudi, da pooblastitveni strežnik odjemalca overi ločeno od uporabnika, ko ta zaprosi za žeton.

2. Implicitno (ang. *Implicit*)

   Ta način se od prvega razlikuje po tem, da pooblastitveni strežnik po overitvi uporabnika nazaj ne vrne pooblastitvene kode, ampak kar žeton za dostop in ta ga nato posreduje do odjemalca. S tem se izognemo ponovnemu dostopu do pooblastitvenega strežnika in tako pospešimo postopek in izboljšamo odzivnost aplikacije. Hkrati pa ta način pooblastitve pomeni, da pooblastitveni strežnik ne more ločeno overiti odjemalca, kar je lahko nevarno.

3. Z uporabo prijavnih podatkov lastnika vira (ang. *Resource Owner Password Credentials*)



Slika 2.7 Potek pooblaščanja s prijavnimi podatki lastnika vira [2].

V tem načinu (Slika 2.7) uporabnik zaupa odjemalcu in mu neposredno posreduje prijavne podatke. Odjemalec jih nato uporabi za pridobitev žetona za dostop pri

pooblastitvenem strežniku. S tem dosežemo to, da odjemalec ne zadrži prijavnih podatkov uporabnika, ampak jih zamenja za anonimen in varen žeton, ki ga lahko potem tudi obnovi, ne da bi uporabnika ponovno prisilil k prijavi. Pooblastitveni strežnik lahko v tem primeru še vedno overi odjemalca, ne pa tudi uporabnika.

4. Z uporabo prijavni podatkov odjemalca (ang. *Client Credentials*)

   V zadnjem načinu uporabnik nima vloge. Odjemalec uporabi lastne prijavne podatke v kakršni koli obliki jih že pooblastitveni strežnik zahteva za pridobitev žetona za dostop. Ta način se uporablja v primeru, da hoče odjemalec dostopati do podatkov, ki si jih lasti sam, torej jih je sam ustvaril in niso last nobenega uporabnika.

OpenID Connect [4][3] je razširitev standarda OAuth 2.0 in je prav tako odprt internetni standard. Namen razširitve je odpraviti pomanjkljivost, prisotno v OAuth 2.0, oziroma dodati uporabno novost odvisno od pogleda. Po standardu OAuth 2.0 lahko odjemalec samo zaprosi za dovoljenje za dostop do zavarovanih podatkov uporabnika, po standardu OpenID Connect pa lahko zaprosi tudi za overitev uporabnika. Na ta način je overitvena in pooblaščalna logika zbrana na enem mestu, ki mu uporabnik zaupa. Poleg omenjene prednosti na tak način dobimo identitetni žeton, ki vsebuje vse nujno potrebne podatke o uporabniku, ti pa so običajno občutljive narave. Primer takih podatkov je seznam vlog uporabnika, na podlagi katerih se aplikacija odloča o pravicah. Takšne informacije morajo biti kar najbolj trenutne, pravilne in skrite. Ker odjemalec zaupa overitvenemu strežniku in mu ta vrne podpisan in šifriran identitetni žeton, lahko odjemalec domneva, da ima pravilne informacije, ki niso prosto vidne. Trenutnost se zagotavlja z omejeno življenjsko dobo žetonov in stalnim obnavljanjem. Ker je OpenID Connect samo razširitev standarda OAuth 2.0, deluje na podoben način in podpira enake načine za pooblastitev.

Poglavje 3 Arhitektura sistema

Okolje, v katerega je sistem postavljen, lahko grobo razdelimo na tri področja, od katerih ima lastne karakteristike. Ta področja so medomrežje, DMZ in notranje omrežje podjetja. Medomrežje je omrežje, ki ni pod kontrolo upravitelja in na njem ni zagotovljena varnost. Notranje omrežje podjetja je načeloma varno in hitro omrežje pod nadzorom upravitelja in je zavarovano s požarno pregrado, prepustno samo za znan promet. DMZ je področje, ki fizično sicer leži v omrežju podjetja, vendar pa je logično odrezano od preostalega notranjega omrežja s požarno pregrado. Ves internetni promet, ki pride do omrežja podjetja, je spuščen le v DMZ, kjer stojijo medomrežju dostopne storitve. Takšna troslojna delitev povečuje stopnjo varnosti v notranjem omrežju podjetjau, kar je zelo pomembno.

3.1 Splošna teorija dobre arhitekture

Dobra arhitektura [5] upošteva lastnosti posameznih področij in jih izkorišča. Pri načrtovanju arhitekture sistema [6] lahko začnemo pri obličju ali pa v zaledju. Če začnemo pri obličju, najprej skiciramo vmesnik, s katerim bo uporabnik delal, in nato zgradimo sistem, ki učinkovito podpira funkcionalnosti tega vmesnika. Če začnemo v zaledju, začnemo z relacijsko-entitetnim modelom podatkovne baze in nato zgradimo sistem, ki omogoča uporabniku prijazno delo s podatkovno bazo. Ne glede na začetek je lahko končni izdelek za uporabnika enak, arhitektura pa različna. V prvem primeru je arhitektura bolj tesno povezana z aplikacijo obličja, v drugem pa s podatkovno bazo.

Kateri pristop je pravi? Če razmislimo, vidimo, da se posamezne komponente sistema spreminjajo z različno hitrostjo. Obličje se vedno spreminja. Lahko imamo namizno, mobilno aplikacijo ali spletno aplikacijo, do katere dostopamo z namizja ali mobilno. Lahko uporabimo stotine ogrodij vsako s svojim slogom programiranja in filozofijo. V primerjavi z obličjem je podatkovna baza veliko bolj stabilna. Tehnologija, filozofija in programski jezik so poznani. Ko je podatkovni model določen, se zelo redko spreminja.

Tako lahko vidimo, da povezanost s podatkovno bazo v bistvu pomeni bolj stabilno in splošno arhitekturo, ki ni podvržena hitro spreminjajočemu se svetu tehnologij obličja. Uporablja funkcionalnosti zaledja, ki so definirane tako, da ni povezave na specifično tehnologijo obličja.

Če torej začnemo v zaledju s podatkovno bazo, najprej definiramo podatkovni model glede na zahteve sistema. Baza mora biti v notranjem omrežju podjetja in poizvedbe do nje prav tako, ker hrani vse podatke in morajo ti biti na varnem. Zaradi varnosti podatkov je dobra ideja, da vnaprej naredimo funkcionalno specifikacijo in z njo definiramo operacije nad podatki v bazi, katere lahko izdelamo z uporabo bazne procedure ali pa ustvarimo aplikacijo, ki deluje kot programski vmesnik za delo s podatkovno bazo. Druga izbira je veliko bolj prilagodljiva, poleg tega pa lahko programiramo v bolj znanem programskem jeziku. Dobra ideja je tudi, da ta aplikacija deluje kot stražar ter overja in pooblašča klicatelje za izvajanje določenih funkcij, saj lahko na tak način poskrbimo, da so podatki dostopni samo ustreznim uporabnikom. Recimo temu vmesniku DBAPI, kar je precej krajše kot programski vmesnik za delo s podatkovno bazo.

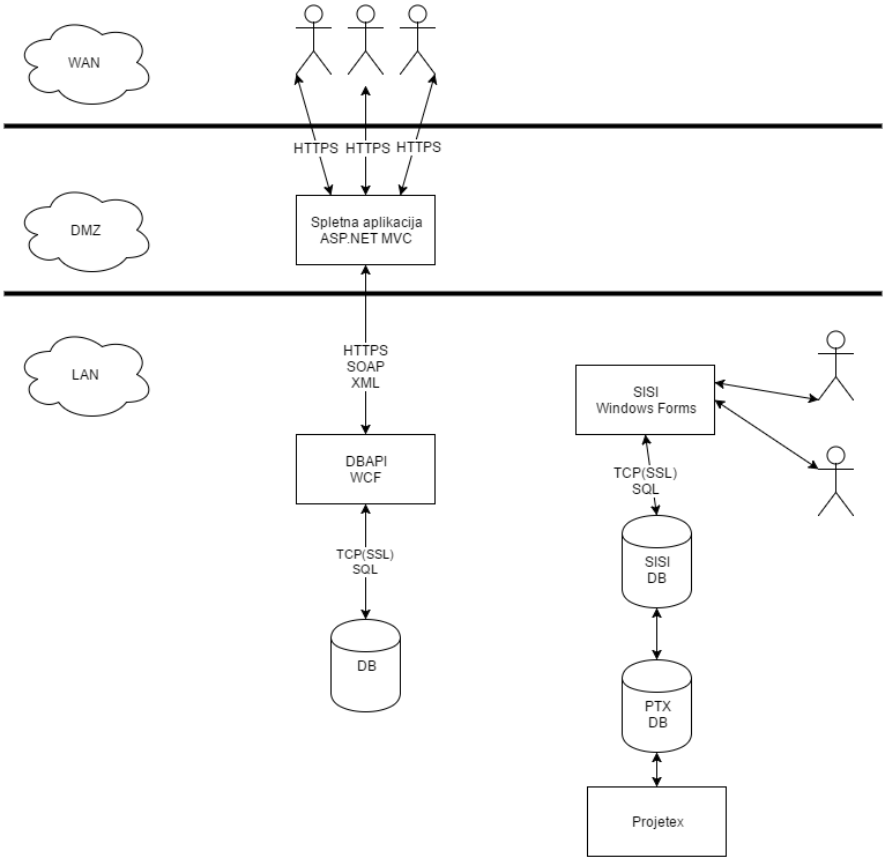
Uporabniki lahko opravljajo zahtevane operacije preko DBAPI-ja, vendar pa tak način dela ni preveč prijazen do uporabnika. Manjka grafični vmesnik, ki delo s programskim vmesnikom predela v privlačno aplikacijo, ki je lahka za uporabo in skriva nepotrebne tehnične podrobnosti komunikacije s programskim vmesnikom

Grobo ocenjeno sta najpopularnejši dve vrsti aplikacij, ki sta namenjeni povprečnemu uporabniku: lokalne in spletne. Lokalna aplikacija se namesti neposredno na napravo in se izvaja na njej. Spletna aplikacija nima namestitve in se izvaja delno na oddaljenem strežniku in delno na uporabnikovi napravi. Lokalna aplikacija je primerna za notranje omrežje podjetja, ker se zaganja na namiznih računalnikih in je pod nadzorom sistemskih upraviteljev. Spletna aplikacija je primerna za DMZ, ker lahko tam do nje dostopajo uporabniki od koder koli in kadar koli.

Recimo, da si želimo vse funkcionalnosti sistema združiti na enem mestu. Če vemo, da bodo uporabniki do sistema dostopali prek medomrežja in notranjega omrežja podjetja, potem izberemo spletno aplikacijo. Če vemo, da bodo uporabniki dostopali izključno iz notranjega omrežja podjetja, potem se moramo vprašati, kaj si želimo od obličja. Lokalna aplikacija je odzivna in učinkovito porazdeli breme sistema, vendar pa je njen grafični vmesnik bolj omejen kot pri spletni aplikaciji zaradi pomanjkanja različnih zunanjih knjižnic, ki obstajajo za spletne strani. Spletna aplikacija centralizira breme sistema in omeji količino procesorskega časa, ki ga lahko dobi posamezni uporabnik, ima pa zelo močna orodja za izdelavo grafičnega vmesnika.

Recimo, da imamo uporabnike iz medomrežja in notranjega omrežja podjetja ter smo izbrali spletno aplikacijo. Spletna aplikacija je umeščena v DMZ, podatkovna baza in DBAPI pa sta v notranjem omrežju podjetja. Spletna aplikacija dostopa do baze prek DBAPI-ja. DBAPI lahko spletni aplikaciji zaupa, da bo ta preverjala uporabnike in njihove pravice ter samo pooblaščene uporabnike napotila do DBAPI-ja. Težava nastane, ker je DMZ odprt za internetni promet in je tam večja možnost vdora. Če pride do tega, lahko napadalec po želji kliče funkcije DBAPI-ja, kar pomeni, da lahko spreminja podatke kogar koli v bazi. Edina omejitev je, da lahko uporablja samo funkcije DBAPI-ja. Dvakratno overjanje in pooblaščanje bi pomenilo nepotrebno delo in bi naredilo obe komponenti bolj kompleksni, zato je naravna odločitev, da overjanje in pooblaščanje centraliziramo v novi komponenti. Ta vsakega uporabnika preveri: če je overjen, njegove podatke spravi v žeton, ki postane edini vir identitete uporabnika za spletno aplikacijo in DBAPI. Na podlagi podatkov v žetonu se komponenti odločata o pravicah uporabnika.

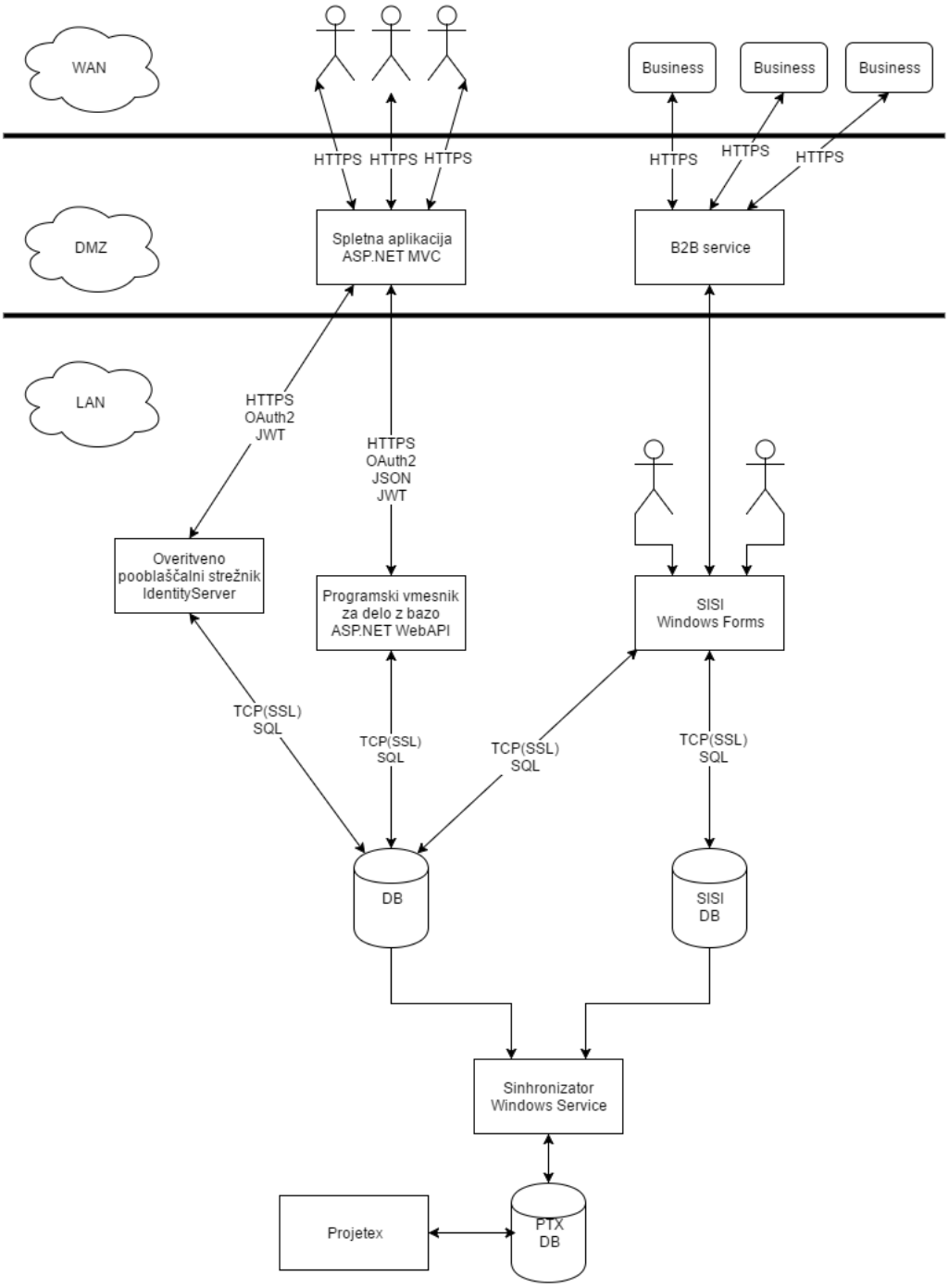
3.2 Obstoječa arhitektura



Slika 3.1 Diagram arhitekture, kot je bila postavljena.

Kot vidimo ta arhitektura (Slika 3.1) ne sledi ugotovitvam iz prejšnjega podpoglavja. Overjanje in pooblaščanje se izvajata tako na spletni aplikaciji kot na DBAPI programskem vmesniku in ni centralizirano v ločeni komponenti. Konkretne posledice takšne zasnove so opisane v podpoglavju o odzivnosti sistema.

3.3 Načrtovana arhitektura

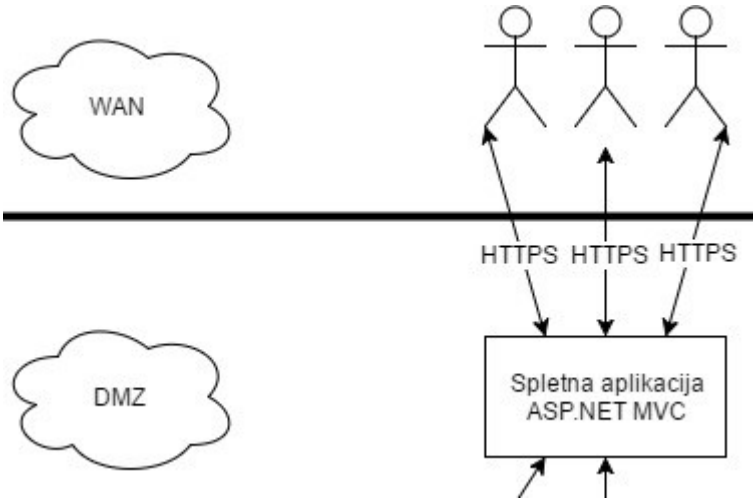


Slika 3.2 Diagram arhitekture celotnega sistema.

Načrtovana arhitektura sistema (Slika 3.2) je precej bližja idealni. Overjanje in pooblaščanje se izvaja v IdentityServer komponenti, ki ustvarja varne žetone. Te potem spletna aplikacija in DBAPI uporabljata za odločitve o pooblaščanju uporabnika. Takšna zasnova poviša tako odzivnost kot povečljivost, zato je obravnavana tam. Posamezne komponente iz zgornje slike bodo opisane v nadaljevanju tega poglavja.

3.4 Jedrni sistem

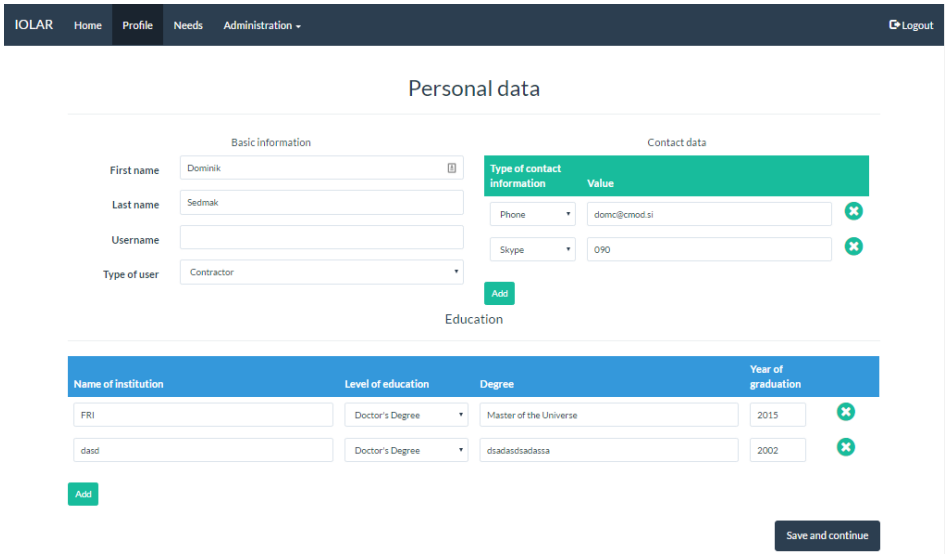
3.4.1 MVC obličje – spletna aplikacija



Slika 3.3 Postavitev spletne aplikacije v celotni arhitekturi.

Namen spletne aplikacije (Slika 3.3) je omogočanje učinkovitega upravljanja s človeškimi viri [7]. Zato jo uporabljajo predvsem zunanji izvajalci (viri) in upravljavci z viri. Izvajalci lahko ustvarijo svoj profil in urejajo svoje podatke. Prek spletne aplikacije prav tako opravljajo testiranje izvajalcev.

Profil se za boljšo uporabniško izkušnjo izpolnjuje v štirih korakih. Prvi (Slika 3.4) trije so odprti takoj po prijavi, zadnji pa šele, ko je vir prestal test in so se v podjetju odločili za začetek sodelovanja. V profilu od uporabnika pridobimo osnovne informacije o njem, njegovi izobrazbi, njegovih izkušnjah s področja prevajanja in prevzamemo kakršne koli datoteke, ki bi jih uporabnik rad priložil. Profil je zasnovan tako, da zamenja CV, ki je samo nestrukturirano besedilo, z grafičnim vmesnikom, ki poskrbi, da so podatki uporabnika pravilno oblikovani in strukturirani. Upravljavci z viri imajo skozi spletno aplikacijo pregled nad ustvarjenimi profili in jih lahko tudi spreminjajo. To pride prav, ko je treba kakšno spremembo narediti v imenu uporabnika. Spletna aplikacija za upravljavce z viri vsebuje iskalnik, prek katerega lahko ustvarjajo kompleksne poizvedbe in preiskujejo profile.



Slika 3.4 Prvi korak izdelave profila.

Glede na podatke, ki so jih kandidati zapisali v profilu, se po potrebi izvede testiranje, s katerim je določena stopnja znanja in sposobnost kandidata. Testi so pripravljeni v spletni aplikaciji glede na kombinacijo jezikovnega para (iz jezika A v jezik B) in teme tako, da je test čim bolj realističen in informativen. Testi sami so v obliki Word dokumentov, njihov prevod pa se ne izvaja v spletni aplikaciji, so pa tu izvedeni drugi koraki procesa. Vir ima preko spletne aplikacije pregled nad potekom testiranja. Vidi lahko, kateri testi ga čakajo, test lahko prevzame in ga tako začne, pri čemer se v aplikaciji spremlja čas reševanja. Ko test reši, ga odda nazaj prek spletne aplikacije, kjer po pregledu testa (prevoda) vidi tudi svoje rezultate. Upravljavci z viri celoten postopek vodijo prek spletne aplikacije.

Spletna aplikacija je zgrajena na osnovi ogrodja ASP.NET Core in gostuje v spletnem strežniku IIS v DMZ-ju. Za overitev uporabnika se zanaša na OP-strežnik, od katerega pri prijavi uporabnika zahteva žeton identitete, žeton za dostop do DBAPI-ja in obnovitveni žeton. Žetone shrani v piškotek in prijavi uporabnika. Nadaljnje odločitve glede pooblaščanja opravlja na podlagi uporabnikovih pravic, ki so shranjene v žetonu identitete. Ko uporabnikova akcija sproži metodo, v kateri se kliče DBAPI, spletna aplikacija iz piškotka pridobi žeton za dostop in ga skupaj s samo http-zahtevo pošlje DBAPI-ju.

3.4.2 WebAPI zaledje – DBAPI

Slika 3.5 Postavitev DBAPI-ja v celotni arhitekturi.

Druga najpomembnejša aplikacija v sistemu je aplikacija, ki nadzoruje in omejuje delo s podatkovno bazo. Programski vmesnik za dostop do podatkovne baze oz. DBAPI (Slika 3.5) gostuje v spletnem strežniku znotraj notranjem omrežju podjetja in ni dostopen iz medomrežja predvsem iz varnostnih razlogov. DBAPI je zgrajen na podlagi ASP.NET WebAPI tehnologije in deluje prek HTTP-protokola. Podatki, ki prihajajo do njega, in tisti, ki jih pošilja, so vsi pozaporedeni v JSON format.

Namen DBAPI-ja je, da varuje podatkovno bazo in da preprečuje preveč tesno zanašanje na specifične tehnologije, kot je MSSQL Server. DBAPI je v bistvu program, sestavljen iz objektov in funkcij ter funkcionalnosti, ki omogoči, da se te funkcije zaženejo po prejemu pravilno naslovljene HTTP zahteve in da se vhodni parametri funkcije pravilno razzaporedeni iz telesa HTTP zahteve. V funkciji lahko potem vhodne podatke preverimo in po potrebi pretvorimo ter nazadnje shranimo v podatkovno bazo, če je bila HTTP zahteva tipa UPDATE, PUT ali DELETE. Če do DBAPI–ja pride zahteva tipa GET, poskrbimo, da ne pride do

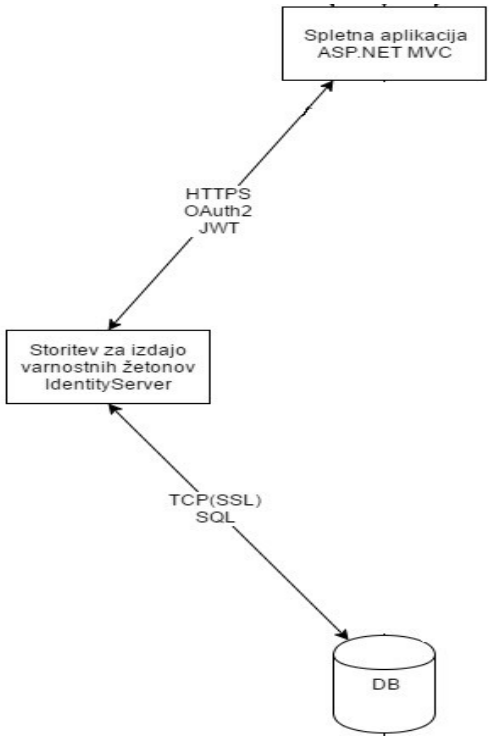
sprememb v bazi, razen za potrebe poznejše revizije (ang. *auditing*). S tem zagotovimo, da zahteve po podatkih nimajo stranskih učinkov.

Preden se zažene funkcija, se seveda preveri, ali je klicatelj pooblaščen za izvajanje te funkcije. Klicatelj mora klicu priložiti žeton za dostop in žeton identitete, če kliče v imenu uporabnika. Oba žetona se najprej preverita: če je bila zagotovljena integriteta (nihče razen PO strežnika ni spreminjal vsebine), potem upoštevamo vsebine žetonov. Pooblaščanje v okviru ASP.NET ogrodja deluje prek »trditev« in te trditve vsebujejo vloge klicatelja. Če klicatelj pripada vlogi, ki je zahtevana za izvajanje funkcije, potem je to dovoljeno.

Katere funkcije DBAPI ponuja, v veliki meri določa obličje. Trenutna različica DBAPI-ja vsebuje administrativne funkcije za upravljanje z vlogami, upravljanje z elementi uporabniškega vmesnika (besedilo, ki se pojavi na spletni aplikaciji), upravljanje z računi in profili uporabnikov ter testi. Račun je ustvarjen, ko se uporabnik prijavi, profil pa, ko uporabnik izpolni podatke na obrazcu v spletni aplikaciji po registraciji, zato so tudi funkcije za upravljanje različne, npr. `Register`, `CheckCredentials`, `ResetPassword`, `ConfirmPasswordToken`, `ConfirmAccount` za upravljanje z računi in `GetPersonalData`, `SavePersonalData`, `GetTechnicalData`, `SaveTechnicalData` za upravljanje s profili.

DBAPI je seveda prilagodljiv in gotovo bodo sčasoma dodane nove funkcije, ki bodo podpirale funkcionalnosti v obličju.

3.4.3 IdentityServer OP-strežnik



Slika 3.6 Postavitev overitveno-pooblaščevalnega strežnika v celotni arhitekturi

Ključni del vsakega sistema, ki uporablja protokol OAuth 2.0, je pooblaščevalni strežnik (Slika 3.6). V primeru sistema v podjetju Iolar smo se odločili, da bomo poleg OAuth 2.0 uporabili še OpenID Connect protokol in na združljiv in fleksibilen način rešili še overjanje uporabnika. Ker je izgradnja strežnika za overjanje in pooblaščanje, ki sledi standardom, konkreten zalogaj, ki ne bi pomenil posebne dodane vrednosti v primerjavi z obstoječimi rešitvami, smo se odločili, da uporabimo pripravljeno rešitev. Izbrali smo IdentityServer [8][7], ker je odprtokoden, zastonj in grajen na ASP.NET MVC tehnologiji in se zato preprosto integrira v celoten sistem, ki ga gradimo. Poleg tega IdentityServer implementira oba omenjena internetna standarda in tudi njune neobvezne razširitve. Ima tudi dobro napisano dokumentacijo in veliko primerov, kar je zelo pomembno.

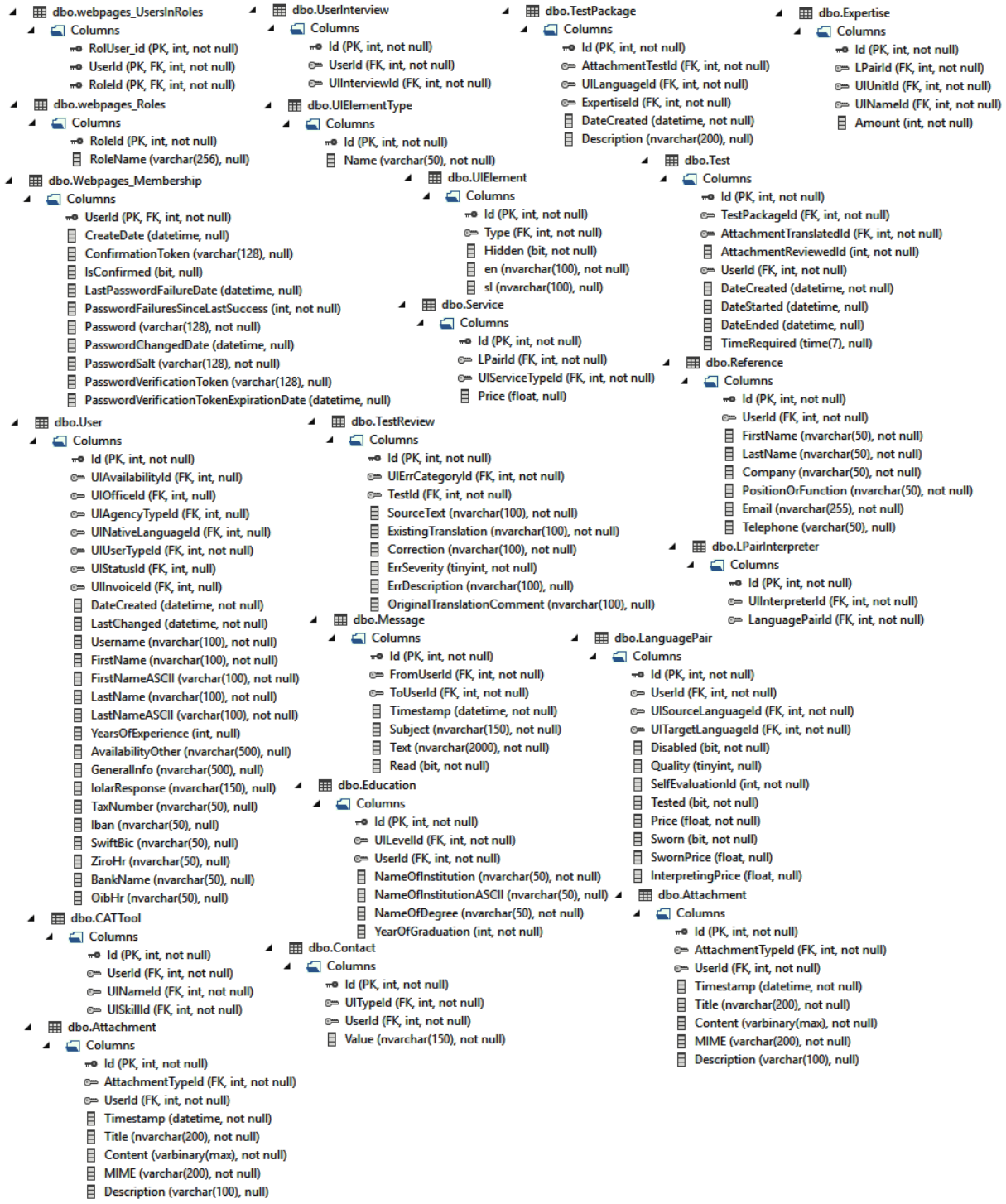
IdentityServer podpira vse štiri načine pooblaščanja, definirane v OAuth 2.0 specifikaciji, vendar pa se v sistemu uporablja način »Prijavni podatki lastnika podatkov«. Za ta način smo se odločili zato, da lahko OP-strežnik ohranimo v notranjem omrežju podjetja in ga tako zaščitimo. OP-strežnik je zelo pomembna komponenta sistema in uspešen napad nanjo bi napadalcem dovolil dostop do podatkov uporabnikov. Tako noben uporabnik nima dostopa do OP-strežnika in spletna aplikacija v njihovem imenu zaprosi za žetone.

Žetone, ki jih proizvaja IdentityServer, za zdaj uporabljata spletna aplikacija in DBAPI. DBAPI in spletna aplikacija zahtevata in uporabljata žeton za dostop in žeton identitete. DBAPI-ju oba posreduje spletna aplikacija v imenu uporabnika oz. ko je to potrebno v svojem imenu. Spletna aplikacija pa oba pridobi sama.

Ker je žeton edini vir identitete in dovoljenj uporabnika, žetonov ne smemo pustiti veljavnih predolgo. Če nekemu uporabniku odvzamemo pravice, se bo to pokazalo šele, ko se bo ta ponovno prijavil, kar se v najslabšem primeru zgodi, ko poteče žeton. Če pa znižamo čas veljavnosti žetona, se bo uporabnik primoran znova prijaviti, kar uporabniku ni preveč prijazno. IdentityServer podpira uporabo obnovljivih žetonov, katerih namen je razrešiti to težavo. Obnovitveni žeton je posebna vrsta žetona, ki nam omogoča, da izdamo nov žeton identitete oz. žeton za dostop, ne da bi uporabnik moral podati prijavne podatke. Njihovo delovanje in oblika sta podrobneje predstavljena v naslednjem poglavju.

3.4.4 Strežnik podatkovne baze

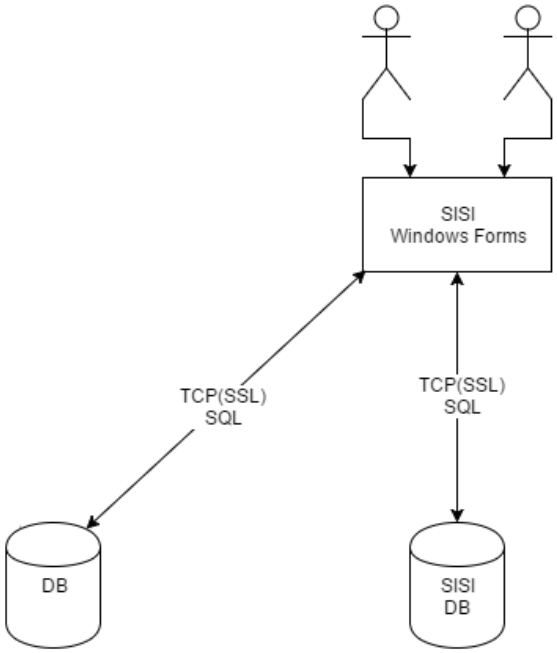
V podatkovni bazi so shranjeni vsi podatki o uporabnikih in vsi podatki, potrebni za delovanje aplikacije, ki ne pripadajo uporabnikom. To so predvsem elementi uporabniškega vmesnika. Načrtovanje relacijsko-entitetnega modela (Slika 3.7) baze ne spada pod arhitekturo sistema, so pa baza sama in njena odzivnost in zmogljivost izredno pomemben del sistema.



Slika 3.7 Relacijsko entitetni model podatkovne baze.

3.5 Širša arhitektura sistema

3.5.1 Podpora projektnemu vodenju



Slika 3.8 Postavitev SISI v celotni arhitekturi

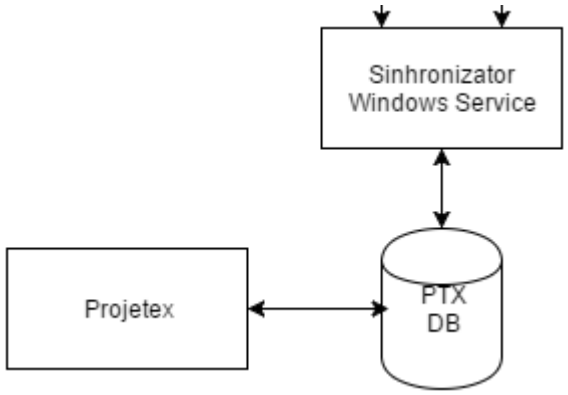
SISI je ime za namizno aplikacijo v razvoju v podjetju Iolar. Aplikacija ni tako tesno povezana v sistem kot preostale, ker je namenjena lajšanju dela projektnih vodij, preostanek sistema je namenjen upravljavcem z viri. Ker pa se delo obeh povezuje, je v sistem povezana tudi aplikacija.

Za potrebe opisa delovanja SISI primarno podatkovno bazo poimenujemo iProki (Slika 3.8). SISI ima trenutno svojo podatkovno bazo, ker pa je eden od ciljev racionalizacije procesov v podjetju ta, da so vsi podatki zbrani na enem mestu v eni obliki, so vsi podatki o izvajalcih po novem in primarno v podatkovni bazi iProki. Zato ima SISI poleg svoje podatkovne baze dostop tudi do baze iProki prek neposredne povezave.

Odločili smo se, da SISI omogočimo neposreden dostop do podatkovne baze namesto prek DBAPI-ja. Za to je več razlogov. SISI je namizna aplikacija, ki se zanaša na to, da je pognana v računalniku z domenskim dostopom, torej takšnim, ki je nameščen na notranje omrežje podjetja. Zato je verjetnost napada veliko nižja kot pri spletni aplikaciji, ki se izvaja v DMZ-ju. Če bi želeli, da SISI uporablja DBAPI za dostope do baze, bi morali spremeniti tako DBAPI

kot tudi aplikacijo SISI, da bi to podpirala. Ker je odločitev o prihodnosti aplikacije pod vprašajem, se tega (še) nismo lotili.

3.5.2 Projetex



Slika 3.9 Projetex v celotni arhitekturi

Projetex (Slika 3.9) je programska oprema podjetja Advanced International Translations in se uporablja v podjetju IOLAR. Pred začetkom projektov iProki in SISI je bil Projetex glavna aplikacija v uporabi. Projetex je lastniški program, kar pomeni, da je njegovo delovanje zaklenjeno ter izključuje prilagoditev zahtevam naročnika. Program ima lastno podatkovno bazo, do katere je omogočen omejen dostop prek programskega vmesnika ODBC. Projetex se uporablja za hranjenje računovodskih podatkov, kar ni načrtovana funkcionalnost spletne aplikacije oziroma SISI, zato se bo uporabljal še naprej. Do nadaljnjega je treba določene podatke sinhronizirati s Projetexovo podatkovno bazo, kar bo opravljala temu prilagojena storitev Windows, ki pa še ni zgrajena.

3.6 Overjanje in pooblaščanje

Overjanje in pooblaščanje sta zelo pomembna procesa v arhitekturi sistema. Odločitve o pooblaščanju izvajajo posamezne aplikacije sistema na podlagi vsebine žetonov, ki so jim posredovani. Najprej je treba žetone pridobiti od OP-strežnika in jih nekje shraniti, nato pa posredovati viru, do katerega želimo dostopati.

3.6.1 Pridobitev žetona

Kakršna koli komunikacija, katere namen je pridobitev žetona, vedno poteka med OP-strežnikom in komponento, ki zahteva žeton. V sistemu, kot je predviden, lahko žeton zahteva samo spletna aplikacija. Za pridobitev žetona uporablja način »Prijavni podatki lastnika

podatkov«. Spletna aplikacija gostuje v spletnem strežniku IIS in teče pod računom, ki ji ga dodeli IIS. Zato ne moremo iz podatkov o računu sklepati o identiteti uporabnika in potrebni so prijavni podatki uporabnika, da lahko aplikacija zahteva žeton v njegovem imenu. Pozneje se lahko dodajo novi odjemalci.

Komunikacija z IdentityServerjem poteka izključno prek protokola HTTP in mora biti zaščitena s SSL ali TLS. Posledično se uporablja.NET razred HttpClient, ki omogoča aplikacijam pošiljanje in sprejemanje HTTP zahtev. IdentityServer razred razširi z dodatnimi metodami, ki olajšajo komunikacijo. V ozadju se na OP-strežnik po standardu pošlje HTTP GET zahteva, s parametri, ki specificirajo najmanj vir, do katerega želijo dostopati (ang. *scope*) in unikatni identifikator odjemalca, ki mora biti registriran pri OP-strežniku. V konkretnem primeru načina »Prijavni podatki lastnika podatkov« se pošljeta tudi uporabniško ime in geslo uporabnika.

Če je overjanje uspešno, OP-strežnik vrne žeton za dostop in žeton identitete ter žeton za obnovitev odvisno od področij, zahtevanih pri poslani zahtevi. Vsebina žetonov je podpisana in šifrirana ter vrnjena kot niz. Odjemalec sam ne more prebrati žetona, razen če je namenjen za njegovo uporabo.

3.6.2 Uporaba žetona

Ta korak vedno poteka med odjemalcem oziroma aplikacijo, ki želi dostop do vira, in strežnikom te dobrine. V sistemu žetone berejo tri aplikacije: OP-strežnik bere žetone za dostop do trditev o uporabniku, DBAPI in spletna aplikacija pa bereta žetone identitete, ker so v njih podatki o vlogah uporabnika.

OP-strežnik ima dostopno točko, kjer lahko z žetonom za dostop pridobimo trditve o uporabniku. Slednja možnost je alternativa temu, da trditve damo kar v žeton identitete. Odločitev za en in drug pristop je pomembna za povezljivost, zato se obravnava tam.

DBAPI in spletna aplikacija morata pri vsaki zahtevi uporabnika znova overiti in pooblastiti, ker je HTTP-protokol brez stanja, zato pri vsaki zahtevi preverita in prebereta žeton ter iz njega razbereta trditve o uporabniku, ki se lahko uporabijo za omejevanje dostopa.

3.7 Prenos podatkov

Drugi tip komunikacije med komponentami sistema je prenos podatkov. Ko je klicatelj overjen in pooblaščen za izvedbo določene funkcije oz. dostop do določenih virov, nastopi prenos podatkov.

Komponente sistema med sabo prenašajo podatke na različne načine. Spletna aplikacija predvsem komunicira z DBAPI-jem. Na tej povezavi se zgodi večina prenosa podatkov med delovanjem sistema. OP-strežnik, DBAPI in SISI dostopajo do podatkovnih baz neposredno.

Med DBAPI-jem in aplikacijami, ki z njim komunicirajo, se podatki prenašajo po protokolu HTTP in so zavarovani s TLS-protokolom. Sami podatki, ki so na obeh straneh transportnega kanala v obliki navadnih objektov, so za potrebe prenosa pozaporedeni v format JSON in na drugi strani razzaporedeni. Poskrbeti moramo tudi, da so vsi objekti jezika, v katerem se programira in ki so pozaporedeni na voljo na obeh straneh komunikacijskega kanala, sicer ne moremo razzaporediti podatkov.

Do zdaj smo vedno govorili o logični postavitvi sistema, vendar pa fizična postavitev ni nujno enaka. Odločitev o fizični postavitvi (kako naj se aplikacije razporedijo po strežnikih) vpliva na omrežne protokole, ki so v uporabi za prenos podatkov. Kot je predvideno, se bo fizična postavitev začela z minimalnim številom strežnikov in potem povečevala glede na potrebe sistema. To pomeni, da imamo lahko vsaj na začetku OP-strežnik, DBAPI in podatkovno bazo v istem strežniku na notranjem omrežju podjetja ter še en strežnik v DMZ-ju za spletno aplikacijo. Takšna fizična postavitev nam omogoča, da izkoristimo zelo močno funkcionalnost, ki jo ponuja SQL Server, in sicer tako podatkovna baza kot tudi aplikacije, ki jo uporabljajo, lahko komunicirajo prek deljenega pomnilnika [9]. V primeru, da preidemo na postavitev s podatkovno bazo na ločenem strežniku, bo kot omrežni protokol uporabljen TCP/IP brez HTTP.

SISI se kot namizna aplikacija ne izvaja v strežniku, zato ne more izkoristiti povezave z deljenim pomnilnikom. Namesto tega uporablja za komunikacijo protokol TCP/IP.

Poglavje 4 Kvalitativne lastnosti sistema

Ob začetku dela na arhitekturi sistema smo si postavili določene cilje oziroma merila, po katerih bi lahko ocenili arhitekturo. Izbrali smo nekaj najpomembnejših in po njih ocenili staro arhitekturo in možne nove. V tem poglavju je predstavljena ocena arhitekture, ki je bila izbrana. V tej oceni sistema je zajeto jedro sistema, ki obsega spletno aplikacijo, DBAPI, podatkovno bazo in OP-strežnik.

Program SISI je del razširjenega sistema, ker mora za del svoje funkcionalnosti dostopati do podatkov o izvajalcih, ti pa so shranjeni v podatkovni bazi, ki je v središču arhitekture sistema. Sam program SISI razvija druga skupina, zato ni bilo mogoče narediti veliko na kvalitativnih lastnostih programa, saj to zahteva večjo kontrolo smeri razvoja aplikacije. Spremembe v arhitekturi sistema hkrati nimajo vpliva na Projetex, saj ta deluje v ozadju in ima le posreden stik s sistemom prek storitve Windows. Projetex je tudi lastniški program in ga ni mogoče spreminjati.

Glavni cilj je bil, da s spremembo arhitekture in kode izboljšamo delovanje jedrnega sistema z vidika v nadaljevanju opisanih kvalitativnih lastnosti. Spremembe so bile narejene v sami kodi, kjer pa to ni bilo mogoče, so bile izvedene s pravilno konfiguracijo. Od komponent sistema je bilo delovanje dveh konkretno spremenjeno in hitrost delovanja dodatnih dveh povišana s pravilno konfiguracijo. Prvi dve sta spletna aplikacija in DBAPI, drugi dve pa podatkovna baza sama in OP-strežnik.

4.1 Odzivnost

Odzivnost sistema okvirno definiramo kot hitrost delovanja z vidika enega uporabnika. V ta namen je bila spremenjena koda aplikacije in DBAPI-ja, poleg tega pa je sama zamenjava tehnologij prispevala k izboljšani odzivnosti.

Čas delovanja katerega koli programa je seštevek časa, ki ga procesor porabi za računanje, in časa, ki ga porabi za čakanje na zaključek IO-operacij. Ko se te zaključijo, so podatki v registrih in računanje se lahko nadaljuje. Že od začetka računalništva je procesorska moč naraščala hitreje kot hitrost pomnilnika, zato današnji programi porabijo več procesorskega časa v čakanju za podatke. Zaradi tega smo se pri izboljšanju odzivnosti sistema osredotočili na operacije oz. klice funkcij, ki izvajajo IO (omrežne, pomnilniške) operacije. V spletni aplikaciji in DBAPI-ju je bilo treba reorganizirati kodo tako, da je število teh operacij kar najmanjše. Za to smo uporabili tehnike, ki so predstavljene v nadaljevanju.

4.1.1 Obstoječa arhitektura – težave

V obstoječi arhitekturi je sistem deloval približno takole: Uporabnik je prek brskalnika z dostopom do spletne strani ustvaril poizvedbo. To je najprej obdelala spletna aplikacija, nato pa je najverjetneje prišlo do klica DBAPI-ja, ki je spet klical podatkovno bazo. Ta je vrnila podatke DBAPI-ju in ta spletni aplikaciji. Zahteve po podatkih so bile preproste, obdelave podatkov pa ni bilo veliko.

DBAPI, zasnovan na CRUD principu: DBAPI je bil sestavljen iz velikega števila preprostih funkcij, ki so približno sledile vzorcu CRUD (ang. *create, read, update, delete*), po katerem so na vsak objekt, ki predstavlja tabelo v podatkovni bazi, definirane štiri osnovne funkcije, ki se lepo prevedejo v SQL-sintakso INSERT, SELECT, UPDATE, DELETE. Rezultat je preprost in »eleganten« vmesnik, ki omogoča vse potrebne operacije za delo s podatkovno bazo. Na žalost preprosto ni vedno najboljše in tudi v tem primeru se izkaže, da tako zasnovan vmesnik sili aplikacijo v urejanje vse poslovne logike in ponuja malo dodatne vrednosti, razen abstrakcije dela s podatkovno bazo. Nekaj testiranja je pokazalo, da je en sam kompleksen klic, ki vrne kompleksen objekt, sestavljen iz vseh potrebnih podatkov za tisto metodo, veliko bolj učinkovit kot več manjših klicev.

Nepotreben prenos podatkov: Po pregledu kode je bilo očitno, da se določene poizvedbe izvajajo izredno pogosto. To so predvsem zahteve po podatkih o elementih uporabniškega vmesnika spletne aplikacije, ki so se morali naložiti za vsako zahtevo HTTP. V obstoječi arhitekturi se je ta poizvedba tudi vsakič izvedla, saj ni bilo izdelanega nobenega

predpomnjenja. Skupaj s preprosto CRUD zasnovo DBAPI-ja je to pomenilo, da se pri vsaki HTTP zahtevi po povezavi prenaša veliko nepotrebnih informacij.

Počasen pozaporejevalnik: Vsi ti podatki so pred pošiljanjem morali biti pozaporedeni in nato na drugi strani razzaporedeni nazaj v C# objekt. Za to nalogo se je uporabljal v WCF vgrajen pozaporejevalnik. Po raziskavi v medomrežju se je izkazalo, da spada med počasnejše in da gre za pozaporejanje znaten del časa, porabljenega za povprečen klic. Obstaja veliko knjižnic, ki bi lahko to operacijo pospešile.

Obsežna sporočila: Format sporočil, ki so se prenašala po povezavi, je bil XML urejen po SOAP-protokolu, kot to zahteva WCF. XML je že sam po sebi precej dolgovezen (velikost pozaporedenega objekta je večja od drugih prenosnih formatov, npr. JSON), SOAP-protokol pa zahteva prenos dodatnih podatkov, ki ne pripadajo objektu, ki se pošilja. Posledično so bila posamezna sporočila precej večja, kot je bilo potrebno za prenos podatkov po povezavi. Manjkalo je tudi stiskanje podatkov pred prenosom, npr. GZIP. Z uporabo GZIP stiskanja bi lahko dosegli veliko zmanjšanje v velikosti poslanih podatkov.

Leno nalaganje: Zadnja težava z vidika odzivnosti, ki se je pokazala v obstoječi arhitekturi, je napačna konfiguracija uporabljenega ORM-ja Entity Framework. Ta je namreč privzeto nastavljen tako, da uporablja t. i. leno nalaganje. Leno nalaganje je funkcionalnost, ki omogoča, da naložimo vrednosti povezanih entitet šele, ko do njih poskusimo dostopati. To zviša hitrost prve pridobitve objekta. Če pa želimo objekt pozaporediti, vnaprej vemo, da bomo morali dostopati do vseh povezanih entitet, kar pomeni, da jih lahko učinkoviteje pridobimo z enim samim klicem.

4.1.2 Načrtovana arhitektura – rešitve

Zdaj, ko imamo pregled nad težavami obstoječe arhitekture, lahko začnemo iskati rešitve. Zato je bila prva optimizacija sprememba zasnove DBAPI-ja v skladu z ugotovitvami o zmanjševanju števila in velikosti IO operacij (Slika 4.1 Slika 4.2). Zaradi te spremembe in pa tudi same spremembe tehnologije (WCF v ASP.NET Core WebAPI) je bilo treba spremeniti tudi kodo spletne aplikacije tako, da je pravilno uporabljala nov vmesnik.

Storitveno usmerjena zasnova DBAPI-ja: V primerjavi s CRUD pristopom je bolj uporabna oblika vmesnika, storitveno usmerjena, kjer vsaka funkcija ponuja neko celovito funkcionalnost oziroma storitev. Poleg funkcionalnosti je v takšnem vmesniku osnovna delitev funkcij še prisotnost trajnih sprememb v bazi. Funkcije, ki vračajo podatke klicatelju, ne smejo spreminjati, dodajati ali brisati podatkov v podatkovni bazi. Takšni funkciji pravimo, da nima stranskih učinkov. Za ostale funkcije pa velja, da ne smejo vračati podatkov iz baze. Z

upoštevanjem teh napotkov dobimo API s precej manjšim številom funkcij, ki pa še vedno omogočajo vse potrebne funkcionalnosti za klicatelje. Tako lahko ena sama funkcija, npr. `SaveProfile(Profile data)`, zamenja ducat drugih, ki posamično posodobijo, dodajo ali izbrišejo eno izmed komponent entitete. Če ima entiteta profil dve povezani entiteti, `Address` in `Education`, potem bi po CRUD vzorcu imeli 9 metod za spreminjanje entitet (ustvarjanje, posodabljanje, brisanje). Če sledimo SOA principu, pa lahko vse izrazimo z eno metodo, ki predstavlja želeno operacijo. Skrajšan primer tega je viden na sliki (Slika 4.1).

```
public void UpdateAddress(Address data)
{
    _iolarDb.Entry(data).State = EntityState.Modified;
    _iolarDb.SaveChanges();
}

public void DeleteEducation(Education data)
{
    _iolarDb.Entry(data).State = EntityState.Deleted;
    _iolarDb.SaveChanges();
}

public class Profile
{
    public Address address { get; set; }
    public EntityState address_state { get; set; }
    public Education education { get; set; }
    public EntityState education_state { get; set; }
}

public void SaveProfile(Profile data)
{
    _iolarDb.Entry(data.address).State = data.address_state;
    _iolarDb.Entry(data.education).State = data.education_state;
    _iolarDb.SaveChanges();
}
```

Slika 4.1 Poenostavljen primer združevanja funkcij vmesnika.

Združevanje klicev podatkovne baze: Tako kot spletna aplikacija kliče DBAPI, koda v vmesniku kliče funkcije objektno relacijskega pretvornika, ki nato v ozadju komunicira s podatkovno bazo. Kot ORM je bil v načrtovani arhitekturi uporabljen Entity Framework Core, kar pomeni, da je bil vmesnik za delo z bazo poznan [10]. EF nam omogoča, da skozi njen vmesnik sestavljamo in izvajamo poljubne poizvedbe SQL, ne da bi uporabljali sam SQL. Njena najpomembnejša funkcionalnost, kot ORM pa je pretvorba C# objektov v pravilen SQL format in obratno. Vmesnik za delo z bazo, ki ga ponuja EF, omogoča vračanje povezanih entitet v enem klicu z uporabo stikov SQL-a, ampak zaradi obstoječe zasnove DBAPI-ja

(CRUD) ta ni bila v uporabi. S premikom na zasnovo po SOA principu so bile te ločene poizvedbe premaknjene v eno skupno funkcijo in smo jih tako lahko združili v en sam klic (Slika 4.2). To je zelo zmanjšalo število omrežnih operacij in izboljšalo odzivnost in hitrost sistema.

```
public User GetUser(int id)
{
    User u = _iolarDb.User.Find(id);
    List<Address> ad = _iolarDb.Address.
        Where(a => a.UserId == id).
        ToList();
    u.Addresses = ad;
    return u;
}

public User GetUser2(int id)
{
    User u = _iolarDb.User.
        Include(b=>b.Addresses).
        FirstOrDefault(a=>a.Id == id);
    return u;
}
```

Slika 4.2 Primer združevanja klicev v podatkovne baze.

Predpomnjenje: Druga večja izboljšava je bila vpeljava predpomnjenja (Slika 4.3) v delovanje spletne aplikacije. C# oziroma kar .NET platforma ponuja implementacijo predpomnilnika z razredom `MemoryCache` in deluje kot preprost slovar. Okrog razreda `MemoryCache` je bilo treba narediti ovoj, ki skrbi za pravilno dostopanje do slovarja in shranjevanje vanj. S to spremembo je ob vsaki zahtevi treba naložiti le podatke, ki so vezani na uporabnika. Podatki o elementih uporabniškega vmesnika se uporabljajo na strežniški in odjemalčevi strani. Na strani strežnika jih lahko pustimo v obliki .NET objektov, za uporabo na odjemalčevi strani pa jih pretvorimo v JSON obliko. V predpomnilnik shranimo kar oba, saj bi morali sicer pri vsaki zahtevi pretvarjati podatke v JSON.

```
public static string UiDictJson
{
    get
    {
        var dict = MemoryCache.Default.Get("UiDictJson") as string;
        if(dict != null)
            return dict;
        lock(cacheLock)
        {
            dict = MemoryCache.Default.Get("UiDictJson") as string;
            if(dict != null)
                return dict;
            dict = JsonConvert.SerializeObject(UiDict);
            CacheItemPolicy cip = new CacheItemPolicy(){
                SlidingExpiration = new TimeSpan(1,0,0)
            };
            MemoryCache.Default.Set("UiDictJson",dict,cip);
            return dict;
        }
    }
}
```

Slika 4.3 Primer uporabe predpomnjenja.

Zgodnje nalaganje: Ena od težav obstoječe arhitekture je bila napačna konfiguracija EF ORM-ja. Vklopljeno je bilo leno nalaganje. V novem DBAPI-ju smo izključili leno nalaganje, kar je tudi merljivo pripomoglo k povišani odzivnosti sistema. Namesto tega smo uporabili t. i. zgodnje nalaganje. V tem načinu eksplicitno povemo EF, naj naloži tudi povezano entiteto kar v eni poizvedbi do podatkovne baze.

Nov pozaporejevalnik: Velik vpliv na odzivnost sistema ima tudi pozaporedenje podatkov v tekstovno obliko in seveda razzaporedenje na drugi strani komunikacijskega kanala, zato je bila izbira pozaporejevalnika precej pomembna. ASP.NET WebAPI ponuja vgrajen pozaporejevalnik Newtonsoft JSON.NET. V primerjavi z WCF-jevim vgrajenim pozaporejevalnikom je JSON.NET hitrejši tudi do 5x pri pozaporedenju in 2x pri razzaporedenju. Obstajajo programske rešitve, ki so še hitrejše od JSON.NET, vendar ima ta pred njimi zelo pomembno prednost, in sicer da je sposobna razrešiti in pozaporediti cikle v podatkovnih strukturah. To je pomembno zaradi načina delovanja Entity Framework. Če ta zazna tuj ključ v tabeli podatkovne baze, v entiteti, ki tabelo predstavlja, ustvari povezavo do entitete, ki predstavlja povezano tabelo, hkrati pa v tej ustvari povezavo nazaj. Ko se podatki naložijo iz podatkovne baze, EF v podatkovni strukturi ustvari cikle, kar spotakne marsikateri pozaporejevalnik.

Če iz podatkovne baze naložimo entiteto `User`, ta pa ima povezano entiteto `Education`, mora po zahtevah EF-ja tudi `Education` imeti povezavo na `User` entiteto. Če v enem klicu naložimo tako `User` kot tudi povezano `Education` entiteto, bo ta imela povezavo (ang. *reference*) nazaj na `User` entiteto, ta pa na `Education`. Običajno pozaporejevalniki takšnih položajev ne dovoljujejo in se ujamejo v cikel, ko poskušajo slediti povezavam v podatkovni strukturi. JSON.NET to rešuje tako, da preverja vse povezave. Če se je neka povezava prej že pojavila po drevesu, ne nadaljuje, ampak samo zapiše povezavo.

Nov protokol in format za prenos podatkov: V obstoječi arhitekturi se je kot glavni prenosni format uporabljal XML z dodatnimi podatki, kot to zahteva SOAP-protokol. S prehodom na ASP.NET WebAPI ogrodje, ki privzeto deluje prek HTTP-protokola in za komunikacijo uporablja slog REST ter JSON kot format za prenos podatkov, se je povprečna velikost sporočil zelo zmanjšala [11]. Manjša velikost sporočila pomeni manj časa, potrebnega za pozaporedenje in razzaporedenje. Poleg te spremembe je novost še to, da je zdaj vključeno stiskanje podatkov GZIP, kar še dodatno zmanjša čas prenosa podatkov. Skupaj so te spremembe pripomogle k okrog 90-odstotnemu zmanjšanju velikosti povprečnega sporočila.

4.2 Povečljivost

Povečljivost (ang. *scalability*) je mera obnašanja in odzivnosti sistema pri rastočem številu uporabnikov. Že iz definicije lahko sklepamo, da je povečljivost do neke mere povezana z odzivnostjo. Če zmanjšamo količino časa, ki ga sistem porabi na povprečnem uporabniku, hkrati povečamo število uporabnikov, ki lahko sistem hkrati uporabljajo. Po drugi strani pa rešitev vprašanja povečljivosti zahteva posebne prijeme, ki nimajo vpliva na odzivnost oz. hitrost aplikacije, so pa izredno pomembni pri scenarijih z veliko uporabniki.

Verjetno najpomembnejši od teh je uporaba asinhronega delovanja (Slika 4.4). ASP.NET že dolgo podpira asinhrono izvajanje, vendar pa je to šele pred kratkim postalo res uporabno in preprosto za implementacijo z uvedbo novega načina programiranja asinhronih funkcij, t.i. asinhrono/čakaj oznak (ang. async/await).

Bistvo asinhronega izvajanja je, da nikoli ne puščamo blokiranih niti. Če katera koli nit pride v stanje, v katerem mora čakati na konec neke operacije, jo sprostimo in uporabimo za drugo delo. Programer mora takšne operacije predvideti in jih ustrezno označiti, da lahko program deluje asinhrono. Vsako funkcijo, kjer bi želeli uporabiti funkcionalnost asinhronega izvajanja, označimo kot asinhrono. Vsako funkcijo, za katero predvidevamo, da bo blokirala, moramo klicati na asinhron način in jo dodatno opremiti z oznako čakaj (ang. await) (Slika 4.4). IIS spletni strežnik podobno kot Apache vsaki zahtevi dodeli svojo nit, kar lahko privede do

pomanjkanja prostih niti, če te pustimo blokirati. Zato je asinhrono izvajanje tu še posebej pomembno.

V primeru enega uporabnika ne dosežemo veliko, tudi če klic funkcije označimo za asinhronega, se delo ne bo zaključilo nič prej, saj po novem prosta nit nima česa početi. Če pa je uporabnikov veliko potem, lahko sproščena nit poskrbi, da vsi dobijo del procesorskega časa, čeprav je zahtev morda več kot je na voljo niti.

Poleg asinhronega izvajanja je za povečljivost zelo pomembna še ena zasnova. To je brezstanjsko delovanje (ang. *stateless*). Asinhrono izvajanje vpliva na uporabo procesorja, medtem ko brezstanjsko delovanje vpliva na uporabo pomnilnika. Če sistem deluje brezstanjsko, to pomeni, da je vsaka zahteva oz. klic metode neodvisna od drugih. Drugače rečeno: vsaka zahteva oz. klic mora vsebovati vse potrebne informacije za njeno uspešno izvedbo.

4.2.1 Obstoječa arhitektura – težave

V obstoječi in tudi načrtovani arhitekturi imamo dve vrsti omrežnih operacij: od spletne aplikacije do API-jev na notranjem omrežju podjetja in od API-jev do podatkovne baze. Zapisov na disk ni, zato za te ni treba skrbeti.

Sinhrono izvajanje: V obstoječi arhitekturi za nobeno od teh mrežnih operacij ni bilo uporabljeno asinhrono izvajanje, zato je bilo veliko procesorskega časa zapravljenega za blokiranje. To je hkrati vplivalo na povečljivost sistema in omejevalo največje število hkratnih uporabnikov.

Hranjenje stanja: Druga težava, ki je vplivala na povečljivost obstoječe arhitekture, je dejstvo, da je zasnovana tako, da ohranja stanje. S tem je mišljeno ohranjanje seje v strežniku spletne aplikacije in pa konfiguracija WCF DBAPI-ja tako, da ta ohranja objekt, ustvarjen pri prvem klicu API-ja, kar je nekakšna seja. Na tak način se lahko izognemo potrebi po overitvi uporabnika pri vsaki zahtevi. V obstoječi arhitekturi žetoni niso uporabljeni, zato bi bilo treba z vsakim klicem posredovati prijavne podatke.

V praksi pa je takšna postavitev pomenila težavo. Za vsakega uporabnika sta obstajali dve seji: ena v DBAPI-ju in druga na spletni strani. Prišlo je do neskladij v trajanju sej in znalo se je zgoditi, da ena poteče pred drugo, ker se nista obnavljali ob istih operacijah. Hkrati so bili določeni podatki (seznam pravic) podvojeni. Če se uporabnik iz spletne strani ne odjavi, seja ostane v pomnilniku, kar lahko rešujemo tako, da omejimo trajanje seje, kar pa ni preveč prijazno do uporabnika.

Hkrati pa to pomeni, da je poraba pomnilnika konkretno večja kot v brezstanjskem sistemu, kar se odraža tudi pri tem, da je v tem načinu WCF omejen na precej majhno število hkratnih sej oz. uporabnikov. Na povprečnem strežniku s štirimi jedri je ta omejitev 400 hkratnih sej oz. 72 hkratnih klicev WCF DBAPI-ja. Trenutno se število morda zdi veliko, vendar pa vsekakor tak način delovanja ni preveč dober s stališča povečljivosti.

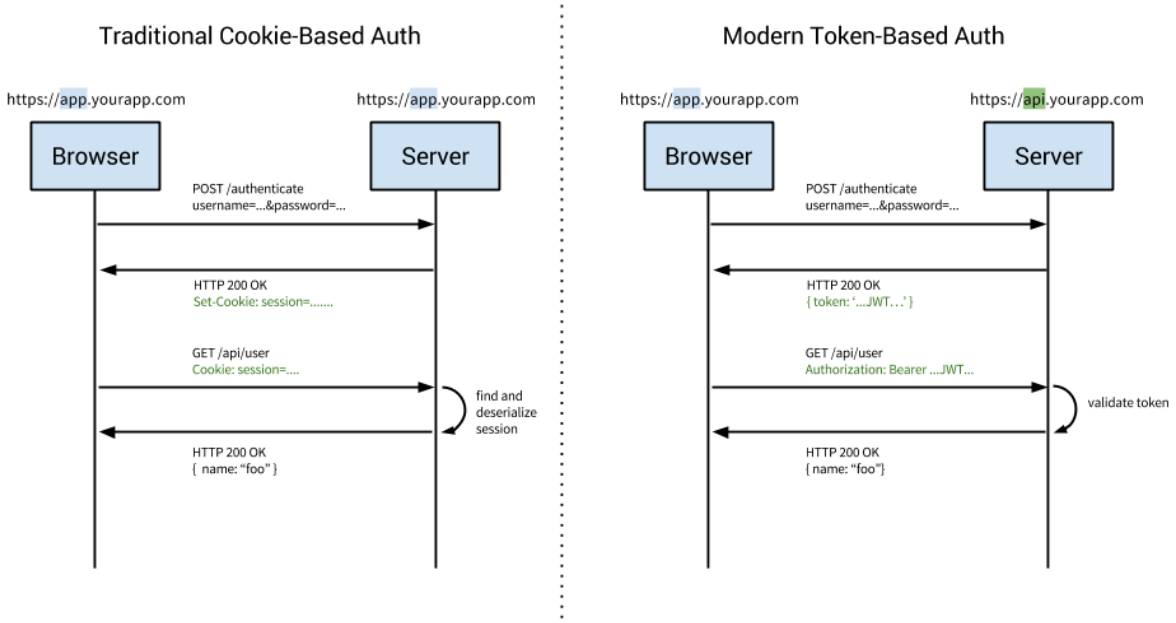
4.2.2 Načrtovana arhitektura – rešitve

Vpeljava asinhronega izvajanja: Ker za DBAPI uporabljamo tehnologijo ASP.NET WebAPI, ki komunicira prek HTTP-protokola, do vmesnika iz spletne strani dostopamo z uporabo razreda `HttpClient`. Ta že sam ponuja asinhrono klicanje: potrebno je samo za asinhrone označiti metode ovoja razreda `HttpClient`. Na strani DBAPI-ja do podatkovne baze dostopamo prek Entity Framework ogrodja. Ta nam omogoča, da metode, ki komunicirajo s podatkovno bazo, kličemo asinhrono ali pa sinhrono. Zato moramo kot asinhrone označiti metode vmesnika in pravilno klicati ponujene asinhrone metode.

```
static async Task CallAPI()
{
    using (var client = new HttpClient())
    {
        client.BaseAddress = new Uri("http://localhost:666/");
        client.DefaultRequestHeaders.Accept.Clear();
        client.DefaultRequestHeaders.Accept.Add(
            new MediaTypeWithQualityHeaderValue("application/json"));
        HttpResponseMessage response = await client.GetAsync("api/data/1");
        if (response.IsSuccessStatusCode)
        {
            Razred ent = await response.Content.ReadAsAsync<Razred>();
        }
    }
}
```

Slika 4.4 Primer uporabe asinhronega klicanja.

Brezstanjska zasnova: Nova arhitektura je bila zaradi težav pri obstoječi arhitekturi zasnovana za brezstanjsko delovanje. Ker je HTTP-protokol, ki ne drži stanja, se za identifikacijo uporabnika prek posameznih zahtev tradicionalno uporabljajo piškotki.



Slika 4.5 Razlika med overjanjem s piškotki in overjanjem z žetoni [2].

Tudi v tem primeru spletna aplikacija uporablja piškotke, vendar na malo drugačen način. Piškotki so namreč uporabljeni samo za to, da brskalnik samodejno z vsako zahtevo pošlje tudi uporabnikov žeton (Slika 4.5) [2]. Piškotka, ki bi identificiral uporabnikovo sejo, ni več, prav tako pa je bila odstranjena celotna seja. DBAPI v tej arhitekturi ne komunicira z brskalnikom, zato potrebujemo neki način, da mu dostavimo uporabnikov žeton. To naredimo tako, da uporabimo Authorization zapis v glavi HTTP zahteve, ko jo pošljemo na DBAPI.

V sistemu se uporabljata dva žetona: žeton identitete in žeton za dostop. Oba v imenu uporabnika od OP-strežnika pridobi spletna aplikacija in ju shrani v piškotek. Žeton identitete vsebuje pravice uporabnika in druge podatke, žeton za dostop pa pooblasti spletno stran, da v imenu uporabnika dostopa do DBAPI-ja. Z vsako HTTP zahtevo tako prideta oba žetona in spletna aplikacija prvega uporabi za odločanje o pooblastitvah uporabnika, prav tako pa ju posreduje vmesniku ob klicu. Ta na enak način preveri pravice uporabnika.

IdentityServer omogoča pridobivanje pravic oz. informacij o uporabniku z žetonom za dostop. Lahko pa tudi vključi informacije neposredno v žeton identitete. Če neke informacije ne potrebujemo prav pri vsaki zahtevi, potem je smiselno te informacije umakniti iz žetona, ker upočasnjujejo razzaporedenje in čas prenosa po mreži. Če pa informacije potrebujemo ob vsaki zahtevi, je bolje, da jih pustimo v žetonu, saj si tako prihranimo klic prek omrežja do OP-

strežnika. V novi arhitekturi so informacije vključene v žeton, ker so te informacije samo pravice uporabnika in so nujne pri vsaki zahtevi.

4.3 Prilagodljivost

Prilagodljivost (ang. *adaptability*) je lastnost sistema, katerega lahko hitro in enostavno spremenimo tako, da ustreza novim zahtevam

Možnost izbire podpornih tehnologij: Ko je Microsoft pretvoril .NET ogrodje v odprtokodni projekt, je kot del tega ogrodja odprtokoden postal tudi ASP.NET. Potem se je Microsoft lotil popolne predelave ASP.NET in .NET ogrodja in rezultat tega dela sta .NET Core in ASP.NET Core. Novo ogrodje poleg Windows okolja deluje tudi na Linux in Mac OS X operacijskih sistemih. Spletna aplikacija in DBAPI temeljita na najnovejši različici ASP.NET ogrodja, torej ASP.NET Core. Seveda to ogromno pomeni za prilagodljivost sistema. Medtem ko smo bili prej omejeni na Microsoftov ekosistem, lahko zdaj izbiramo. V prihodnosti se lahko sistem premakne na drug operacijski sistem in drug strežniški program ter tako prihrani pri licenčnih stroških. IdentityServer je za zdaj še vedno na starem ogrodju ASP.NET, kar pomeni, da celoten sistem še ni pripravljen na zamenjavo okolja, je pa nova različica, ki uporablja ASP.NET Core, v razvoju: ko bo na voljo, se bo sistem premaknil nanjo.

V obstoječi arhitekturi je bila spletna aplikacija zgrajena s starejšo različico ASP.NET ogrodja, DBAPI pa je uporabljal WCF. ASP.NET je imel določeno mero podpore za poganjanje v sistemih Linux in Mac OS X prek ogrodja Mono, vendar ta ni podpiral vse funkcionalnosti. WCF pa je bil povsem vezan na operacijski sistem Windows.

Druga točka, kjer omogočamo prilagodljivost, je v izbiri tehnologije podatkovne baze. Ena od prednosti obsežnejšega ORM-ja, kot je Entity Framework, je, da nam skrije podrobnosti dela z bazo in predstavi enoten vmesnik. Podpora Microsoft SQL Server podatkovni bazi je vključena v Entity Framework, z uporabo razširitev pa je podprtih večina večjih imen v svetu podatkovnih baz. Z lahkoto najdemo razširitve za MySQL, Oracle, SQLite, PostgreSQL itd. Prednosti menjave tehnologije podatkovne baze so enake kot pri menjavi operacijskega sistema.

Neodvisnost komponent sistema: Prilagodljivost sistema se seveda ne meri samo po tem, kakšne tehnologije lahko zamenjamo, ampak tudi po tem, za kaj ga lahko še uporabimo. Komponente sistema so zasnovane kar najbolj neodvisno, zato se lahko uporabijo tudi zunaj trenutno predvidenih vlog. OP-strežnik je knjižnica, tako da ima precej splošno strukturo, kar pomeni, da ga lahko uporabimo zunaj sistema. Trenutno v podjetju ni sistema za overjanje

domenskih uporabnikov zunaj notranjega omrežja podjetja. V prihodnosti bi se lahko OP-strežnik uporabil za to nalogo brez večjih problemov.

Širjenje obličja: Trenutno je spletna aplikacija edini odjemalec DBAPI-ja, vendar pa sistem dovoljuje dodajanje novih. Novi odjemalci morajo biti samo sposobni pridobiti žeton od OP-strežnika in ga posredovati vmesniku. Če DBAPI premaknemo v DMZ in ga odpremo za zunanji promet, lahko v kombinaciji z odprtim OP-strežnikom partnerjem omogočimo samodejno posodabljanje podatkov prek vmesnika.

4.4 Povezljivost

Povezljivost (ang. *interoperability*) lahko definiramo kot sposobnosti komunikacije s starimi in novimi tehnologijami brez večjih težav.

Uporaba široko podprtih tehnologij: Arhitektura sistema je zgrajena na preprostih, široko uporabljanih tehnologijah in arhitekturnih vzorcih. Glavni komunikacijski protokol je HTTP. HTTP je danes izredno široko podprt protokol. Komunikacija poteka v slogu REST, kar pomeni, da lahko vsaka naprava preprosto komunicira s sistemom prek znanih HTTP-zahtev. Format za prenos podatkov je JSON, ki je te dni tako široko podprt kot XML. Poleg tega je delo z njim lažje, ko gre za JavaScript odjemalce. To je pomembno, ker je teh vse več.

OP-strežnik implementira OAuth 2.0 in OpenID Connect internetne standarde. Glede na to, da te standarde uporabljajo veliki uporabniki, kot so Facebook, Google, Microsoft in drugi, smo lahko prepričani, da bodo še kar nekaj časa ostali v veljavi in bo zato povezljivost sistema s prihajajočimi tehnologijami ostala visoka.

Povezljivost s starejšimi tehnologijami: Povezljivost s starejšimi tehnologijami ne bi smela biti problematična. Med starejše ali raje zrele tehnologije lahko štejemo ogrodja za izdelavo namiznih aplikacij, kot je Windows Forms ali pa WPF oz. v svetu Linuxa GTK+ in QT Framework. Microsoft za svoje .NET tehnologije priskrbi razred HttpClient, ki ga je mogoče razširiti z ustreznimi NuGet paketki in tako dobiti delujoč odjemalec, ki lahko komunicira z OP-strežnikom in DBAPI. Povezljivost z aplikacijami, zgrajenimi z GTK+ ali QT, ni tako preprosta, saj nimamo pred pripravljenih rešitev. Vendar pa lahko glede na to, da OP-strežnik deluje po odprtih internetnih standardih, uporabimo katero od že napisanih knjižnic za komunikacijo.

Izboljšave glede na obstoječo arhitekturo: Če primerjamo ta slog komunikacije s tistim v prejšnji arhitekturi, vidimo, da je precej bolj prilagodljiva. Sledenje standardom in uporaba navadnega HTTP-protokola omogoča uporabo za to spisanih knjižnic in nas tako razbremeni

pisanja kode za komunikacijo. WCF za komunikacijo uporablja SOAP-protokol s pred pripravljenimi metapodatki, medtem ko je prenosni format XML. Za starejše tehnologije to ni težava, ker je SOAP bil ustvarjen v času namiznih aplikacij za potrebe poslovnih uporabnikov, pri novejših pa se spotakne. Klicanje SOAP storitev iz brskalniških JavaScript odjemalcev ni preveč udobno oz. je kar slabo podprto. Enako velja za android podporo, tako da menimo, da SOAP komunikacija ne povečuje povezljivosti.

4.5 Varnost

Varnost je od vseh petih zaželenih kvalitativnih lastnosti sistema daleč najpomembnejša. Bila je tudi glavna pobuda za spremembo arhitekture. Ena od nalog sistema je upravljanje s človeškimi viri, zato spletna aplikacija od uporabnika zbira osebne podatkov. Nekateri od teh so javni in neproblematični, drugi pa zasebni in občutljivi, zato jih je treba temeljito zavarovati.

Zaradi pomembnosti varnosti smo se odločili za strukturiran pristop. Varnostna analiza in načrt arhitekture z vidika varnosti sta bila pripravljena na podlagi najpogostejših kategorij groženj po analizi Open Web Application Project (OWASP) [12]. Seveda je mogoče vzeti sistem še bolj pod drobnogled, vendar je za diplomsko delo na voljo omejen čas. Namen je, da v prihodnosti še izboljšamo varnost celotnega sistema.

V nadaljevanju si bomo ogledali najpogostejše kategorije groženj in si pogledali, s kakšnimi prijemi se jim v arhitekturi sistema izogibamo oz. kako se pred njimi branimo.

4.5.1 Vrinjenje

Do vrinjenja (ang. *injection*) lahko pride, če napadalec izkoristi dejstvo, da neka aplikacija njegove vnose neposredno uporabi pri sestavljanju poizvedb. Napad z vrinjenjem ni omejen samo na SQL-poizvedbe, ampak se lahko zgodi, kjer koli se poizvedba do baze pošlje v besedilni obliki. To se lahko zgodi pri poizvedbah do LDAP in Active Directory shramb in drugje. Če napadalcu uspe, lahko dostopa do podatkov, do katerih ni pooblaščen ali pa jih celo spreminja.

V sistemu se napadom vrinjenja izognemo tako, da za vse poizvedbe proti podatkovni bazi uporabljamo parametrizirane klice namesto sestavljenega niza. Parametrizirani klici delujejo tako, da vhodne podatke poizvedbe pošljejo ločeno od samega ukaza SQL in jih nato v podatkovni bazi združijo. To združevanje je narejeno tako, da ne more priti do napada z vrinjenjem. Razvijalec dela s posebnimi LINQ metodami, ki se prevedejo v SQL. Entity

Framework v ozadju sestavi dejansko poizvedbo. Entity Framework nam omogoča tudi neposredno izvajanje SQL poizvedb, vendar se te funkcionalnosti v sistemu ne uporablja.

Tako DBAPI kot tudi OP-strežnik uporabljata Entity Framework za delo s podatkovno bazo, zato nista dovzetna za napade tega tipa.

4.5.2 Napačno zasnovano overjanje in upravljanje s sejo

Cilj napadalca, ki izvaja napad iz te kategorije, je pridobiti začasen ali trajen dostop do uporabniškega računa nekega uporabnika. Skoraj vedno so cilj privilegirani uporabniki z višjo ravnijo pravic kot napadalec.

Ta tip napada se zanaša na napake v overjanju in upravljanju s sejo. Najpogostejše napake, ki omogočajo tak tip napada, so te.

1. Občutljivi podatki, kot je geslo, so v bazi shranjeni v čistopisu.

   Arhitektura sistema poskrbi za pravilno varovanje prijavnih podatkov. Vsako geslo je najprej zgoščeno z JavaScript knjižnico na odjemalčevi strani, preden se sploh pošlje prek medomrežja. Nato je na strani strežnika ponovno zgoščeno pred shranjevanjem. To naredimo zato, ker ne moremo zaupati uporabniku, da ne bo spremenil delovanja skripta pri sebi. Na tak način zagotovimo, da gesla v čistopisu nikoli ne pošljemo preko medomrežja, tudi če je povezava zaščitena.

2. Prijavne podatke je mogoče uganiti ali pa se jih da zaobiti s slabo zasnovanimi metodami za upravljanje z računi.

   Od uporabnika zahtevamo, da si izbere dovolj kompleksno in ne pogosto uporabljeno geslo. To dosežemo tako, da zahtevamo, da ima geslo najmanj 8 črk, med njimi pa mora biti vsaj ena številka in vsaj ena velika črka. Poleg tega skript na odjemalčevi strani geslo preveri s seznamom najpogostejših gesel, tako da smo lahko prepričani, da geslo ni preprosto uganljivo. Sprememba gesla je mogoča samo, ko je uporabnik že prijavljen in tudi takrat zahteva vpis gesla. Ponastavitev gesla deluje tako, da se na e-poštni naslov pošlje povezava, s klikom na katero lahko nastavimo novo geslo. Geslo se ne ponastavi takoj ob začetku postopka, zato nekdo, ki nima dostopa do e-pošte uporabnika, ne more spremeniti gesla.

3. Aplikacija dovoli ponovno uporabo sej z istim ključem po uspešni odjavi uporabnika.

Ta problem je rešen zelo elegantno, in sicer tako, da komponente sistema ne uporabljajo seje. Sistem je zasnovan tako, da deluje brez pomnjenja, kar omogoča visoko povečljivost, varnost pa je dodaten bonus. Vsi podatki, ki identificirajo uporabnika, pridejo skupaj z zahtevo v žetonu identitete.

4. Prijavni in drugi občutljivi podatki se prenašajo prek nezaščitene povezave.

   V procesu razvoja komponente sistema med seboj in z uporabnikom komunicirajo brez zaščite. To je iz čisto praktičnih razlogov, saj upravljanje s certifikati na več napravah ni najpreprostejše. Seveda je vsa koda na mestu za vklop šifriranja. Ko bo sistem postavljen v produkcijsko okolje, bodo vse povezave, tudi tiste znotraj notranjega omrežja podjetja, zaščitene.

4.5.3 XSS-ranljivost

XSS-napad (ang. *cross-site scripting*) je napad na spletno stran z vrivanjem zlonamerne kode, katerega namen je prevzemanje seje uporabnikov, preusmerjanje na druge zlonamerne strani ali kraja podatkov. Ranljivost na XSS-napad je med spletnimi aplikacijami najbolj razširjena. Napad je mogoč, kadar koli spletna aplikacija prikaže vnos uporabnika, ne da bi iz njega najprej odstranila posebne znake. Če je ta vnos prikazan samo uporabniku, ki ga je vpisal, še ni prišlo do uspešnega napada. Za to se morajo vneseni podatki prikazati drugim uporabnikom.

Arhitektura sistema ima dve ravni obrambe pred XSS-napadi:

1. Validacija vseh vnosov uporabnika

   V prvem koraku se preverijo vsi vnosi uporabnika za morebitno zlonamerno kodo. ASP.NET ima vgrajeno samodejno preverjanje vnosov. Če zazna kakršne koli potencialno nevarne znake, na to opozori uporabnika in prekine izvajanje zahteve. Tako je uporabnik oz. v tem primeru napadalec primoran spremeniti svoje vnose, če želi nadaljevati. Je pa tudi res, da ta funkcionalnost ni popolna. V zelo specifičnih primerih se lahko najde način, kako spraviti zlonamerno kodo skozi sito. Zato imamo v sistemu še drugo raven zaščite.

2. Filtriran prikaz vnosov

   Ker je bila spletna aplikacija najprej napisana za staro ASP.NET-ogrodje, za izdelavo pogledov uporablja tehnologija Razor. Novejša tehnologija so t. i. pomožne značke (ang. *tag helpers*), ki se uporabljajo v ASP.NET Core, vendar pa sta podprta oba načina. V Razor pogledih velja, da so vsi izpisi uporabniških podatkov

samodejno HTML kodirani, razen v primeru, da sami eksplicitno zahtevamo drugače. To pomeni, da nikoli ne bo prikazan nekodiran vnos uporabnika. V sistemu se sicer uporablja ena sama zahteva po nekodiranem prikazu podatkov. V tem primeru se podatki ročno kodirajo na strežniški strani pred pošiljanjem odjemalcu, zato lahko rečemo, da sistem ni dovzeten za napade tipa XSS.

4.5.4 Nezavarovane reference na podatke v zaledju

Ta ranljivost je precej pogosta in v najslabšem primeru napadalcu omogoči branje in spreminjanje poljubnih podatkov. Veliko spletnih strani deluje tako, da uporabniku prikažejo uporabniški vmesnik, ki je napolnjen s podatki iz podatkovne baze, uporabnik pa te podatke skozi vmesnik spremeni in pošlje nazaj. V strežniku se potem spremembe shranijo v podatkovno bazo. Spletna aplikacija v uporabniški vmesnik vključi referenco na vnos v bazi, ki je vir podatkov za tisti del vmesnika in hkrati ponor spremenjenih podatkov. Nevarnost tega pristopa je v tem, da lahko premeten uporabnik spremeni te reference in tako spremeni poljubne vnose v podatkovni bazi ali pa jih prebere.

Obstajata dva načina razrešitve te nevarnosti:

1. Posredne reference, vezane na uporabnika

   Takšna rešitev zahteva, da strežnik deluje na način s pomnjenjem, zato ni uporabljena v arhitekturi tega sistema. Deluje pa tako, da v vmesniku namesto neposrednih referenc na vnos v podatkovni bazi uporabimo kakršen koli drug način identifikacije. Ta referenca kaže na vnos v seji uporabnika, kjer je zapisano, kateri vnos v bazi ustreza poslanim podatkom. S tem dosežemo, da uporabnik nikoli ne vidi referenc, hkrati pa preprečimo njihovo spreminjanje, saj se lahko na ta način sprememba podatkov zgodi le, če obstaja vnos v seji, ki pove, kam v bazo spadajo podatki. Tega vnosa seveda ne bo, če je bila posredna referenca spremenjena.

2. Preverjanje pooblastil za delo s podatki

   Ta način se uporablja v tem sistemu, ker ne zahteva seje. V osnovi podelimo uporabniku pravice za spreminjanje in branje svojih podatkov. Vsakič, ko uporabnik poskusi podatke spremeniti, se najprej preveri, ali so podatki res njegovi. Če je poskusil spremeniti reference tako, da bi spremenil neki drug objekt, se operacije zavrne, ker nima dovolj pravic. Isto velja tudi za branje podatkov.

4.5.5 Napačne varnostne nastavitve

To je verjetno najbolj raznovrstna kategorija ranljivosti zaradi količine različnih ogrodij programov in tehnologij, ki so na voljo razvijalcem. Treba je tudi ločevati med nastavitvami v času razvoja in končnimi nastavitvami v produkcijskem okolju. V času razvoja si lahko namreč privoščimo določene izjeme in ne varne nastavitve. Če posplošimo možne rešitve za razna ogrodja in programe, lahko izpostavimo nekaj najpomembnejših.

1. Redne posodobitve

   Vedno je poskrbljeno, da so vse komponente sistema posodobljene na njihove zadnje različice. Po potrebi se izvedejo spremembe v kodi programov, če je sprememba v delovanju ogrodja velika. V določenih primerih posodobitve niso zastonj in niso samo varnostne, npr. nova različica operacijskega sistema. V teh primerih je posodabljanje odvisno od finančnih virov bolj kot od želje po dodatni varnosti, vendar pa je na splošno sistem vedno kar najbolj posodobljen.

2. Odstranitev privzetih računov oz. sprememba njihovih gesel

   Pri razvojnih ogrodjih po navadi ni privzetih računov. Ti pridejo pri zaključeni programski opremi oz. v času razvoja. Konfiguracija operacijskega sistema in spletnega strežnika je v rokah sistemskih upraviteljev, ki poskrbijo za odstranitev takšnih računov. Kar se tiče komponent sistema, pa velja, da bo podatkovna baza izpraznjena pred premikom v produkcijsko okolje, kar pomeni, da bodo odstranjeni tudi vsi testni računi in ostali podatki, ustvarjeni med razvojem.

3. Sprememba privzetih varnostnih nastavitev in upravljanje z razvojnimi okolji

   ASP.NET Core je vpeljal nov način upravljanja z razvojnimi okolji in ga poenostavil. V sistemu sta nastavljeni dve okolji: razvojno in pred produkcijsko. Ko se komponente sistema objavijo v njihovih strežnikih, se samodejno upoštevajo pred produkcijske nastavitve. Ker sistem še ni pripravljen za uporabo oz. izdajo, še ni produkcijskega okolja. Privzete nastavitve so spremenjene v vsakem od okolij.

4. Ustrezno nastavljeno poročanje o napakah

   Končnim uporabnikom seveda ne smemo prikazovati preveč informacij o delovanju sistema, hkrati pa je koristno, če nam lahko sporoči, kaj je bila napaka. Zato so v razvojnem okolju strani za napake polne informacij, v pred produkcijskem pa precej bolj skope.

4.5.6 Manjkajoč nadzor dostopa na nivoju metod

Ta ranljivost je tudi precej pogosta. Za dobro varnost je treba preveriti pravice uporabnika, preden se izvede katera koli metoda v strežniku. Veliko aplikacij preverja pravice uporabnika samo pri sestavljanju pogleda, kjer poskrbijo, da uporabniški vmesnik ne prikazuje tistega, za kar uporabnik ni pooblaščen. Res pa samo to, da funkcionalnosti niso dostopne prek grafičnega vmesnika, še ne pomeni, da sploh niso dostopne. Napadalci lahko komunicirajo s strežnikom, ne da bi sploh uporabljali brskalnik, zato moramo poskrbeti, da se pravice preverjajo pred izvajanjem metod.

V sistemu, kot je bil načrtovan, se vedno preverjajo pravice uporabnika. Vse metode so zavarovane, razen tistih, ki so namenoma puščene odprte, npr. Login, Register … Seveda je poskrbljeno tudi za to, da se uporabniku na spletni strani ne pokažejo vsebine, za katere nima dovolj pravic. Te so pridobljene iz žetona identitete, ta pa je šifriran in popisan tako, da lahko vedno ugotovimo, ali je prišlo do kakršne koli spremembe podatkov v žetonu in tako zagotovimo, da si uporabnik ne pripiše pravic.

V ASP.NET ogrodju pooblaščanje poteka prek atributov na metodah. Z njimi lahko omejimo dostop do metode in na tak način deluje tudi izdelan sistem.

4.5.7 Ponarejanje spletnih zahtev (CSRF)

Pri napadu s ponarejanjem spletnih zahtev napadalec ponaredi sicer legitimno HTTP-zahtevo in uporabnika, ki je v tistem trenutku prijavljen v neko spletno stran, ukani v pošiljanje te zahteve z uporabo drugega napada, kot je XSS, ali pa z uporabo zlonamerne http povezave poslane na e-pošto ali objavljene na kakšni spletni strani oz. družabnem omrežju.. Ker je uporabnik v spletno aplikacijo prijavljen, je za strežnik zahteva videti v redu.

Ta tip napada je kar pogost, vendar pa je na rešitev preprosta. Če program z vsako HTTP-zahtevo, ki zahteva spremembo podatkov (po navadi je zahteva tipa POST), pričakuje enoličen žeton, ki ga je sam ustvaril, potem ponarejena zahteva propade, ker ne more ponarediti žetona.

ASP.NET ogrodje ima vgrajeno metodo za preprečevanje ponarejanja spletnih zahtev. Deluje takole: Odjemalec od strežnika zahteva stran, na kateri je obrazec, ta mu odgovori hkrati, pa v obrazec doda skrito naključno vrednost in to isto vrednost k odgovoru doda kot piškotek. Ko uporabnik odpošlje obrazec, se pošlje skrita vrednost v piškotku in kot del obrazca. Če sta bili obe vrednosti posredovani in sta enaki, je zahteva legitimna. Če manjka ena od vrednosti ali pa nista enaki, potem vemo, da je bila zahteva ponarejena in jo blokiramo. Če napadalec izvede CSRF napad, lahko uporabnika ukane v to, da skupaj z zahtevo pošlje piškotek z žetonom za

preprečevanje. Ne more pa predvideti vrednosti žetona, ker je ta šifriran, zato ne more zahtevi dodati parametra z vrednostjo tega žetona. Rezultat je zahteva s piškotkom, vendar brez parametra z vrednostjo žetona, kar prepoznamo kot napad.

Vse komponente sistema uporabljajo ta sistem za preprečevanje ponarejevanja zahtev.

4.5.8 Nepreverjene preusmeritve in posredovanja

Zelo pogosta funkcionalnost spletnih strani je preusmerjanje na podlagi URL-parametrov. Če uporabnik želi na specifično podstran neke spletne strani, pa ni prijavljen, ga ta lahko preusmeri na prvo stran, kjer se prijavi, nato pa je samodejno preusmerjen nazaj na stran, ki jo je želel obiskati.

Takšna funkcionalnost je dobrodošla s strani uporabnika, vendar hkrati omogoči napad, ki sicer ne bi bil mogoč. Napadalec lahko ustvari URL povezavo, ki uporabnika po uspešni prijavi v aplikacijo preusmeri na zlonamerno spletno stran. To povezavo lahko objavi kjer koli ali pa jo pošlje po e-pošti.

Zlonamerno posredovanje deluje podobno, vendar pa ima drug namen. Napadalec ustvari URL s parametrom, ki pove spletni strani, kam naj ga pošlje po uspešno izvedeni zahtevi. Če nismo pazljivi, lahko na tak način stran omogoči napadalcu dostop do funkcionalnosti in delov vmesnika, za katere nima dovolj pravic. To se lahko zgodi, če pred posredovanjem ne preverimo pravic uporabnika.

Rešitev v obeh primerih je relativno preprosta. Lahko odstranimo oz. ne vgradimo podpore preusmerjanju in posredovanju ali pa samo omejimo preusmeritve in posredovanja na takšna, ki jih ustvari strežnik brez podatkov, ki bi jih podal uporabnik. Arhitektura tega sistema uporablja prvi pristop in uporablja samo preusmeritve, ki so predvidene v kodi aplikacij, zato ta napad ni možen.

Poglavje 5 Sklepne ugotovitve

V okviru diplomskega dela je bil zasnovana in izdelana arhitektura sistema, ki podpira dejavnosti aplikacij obličja. Ta za zdaj obsega spletno aplikacijo za upravljavce z viri in namizno aplikacijo SISI za projektne vodje. Pred začetkom dela smo definirali ciljne kvalitativne lastnosti, ki naj bi jih sistem imel in arhitektura je bila zasnovana v tej smeri. Na koncu smo dobili sistem, ki dosega te cilje, kar pomeni, da je odziven, povečljiv, prilagodljiv, povezljiv in kar je najpomembnejše: varen.

Sistem se za zdaj še ne uporablja v produkcijskem okolju, ker je potrebno še delati na obličju, vendar pa se zdaj lahko delo na to osredotoča.

Izdelava sistema je pomemben mejnik v projektu racionalizacije ključnih procesov podjetja, saj pomeni, da se lahko znova posvetimo funkcionalnostim uporabniku vidnega dela in se pri tem zanašamo na relativno stabilno postavitev podpornih komponent.

Arhitekturo sistema je seveda mogoče še izboljšati. Kar se zadeva varnosti sistema, je mogoče narediti še več. Trenutno je arhitektura zasnovana tako, da minimizira možnost uspešnega napada prek najpogosteje izkoriščenih napadalnih vektorjev. Treba je pregledati dovzetnost za napad po bolj neobičajnih metodah in arhitekturo ustrezno spremeniti. V ta namen je organizacija OWASP pripravila dokument Application Security Verification Standard [13], katerega nasvete bi bilo treba upoštevati.

Izboljšati je mogoče tudi prilagodljivost sistema. IdentityServer je trenutno zgrajen na polnem .NET-ogrodju, tako da se ga ne da prenesti na drug operacijski sistem oziroma vsaj ne s polno zmogljivostjo. To seveda trenutno ni velika težava, vendar pa ga bo v prihodnosti nadomestila novejša različica, ki temelji na ASP.NET Core in je trenutno v izdelavi.

Literatura

[1] »ASP.NET core documentation — ASP.NET documentation,« v ASP.NET Core Documentation, 2016. [Online]. Dosegljivo: https://docs.asp.net/en/latest. Dostopano: 15. avgust 2016.

[2] S. Peyrott, R. Chenkie, and M. Woloski, »Cookies vs tokens: The definitive guide,« v Auth0, Auth0, 2016. [Online]. Dosegljivo: https://auth0.com/blog/cookies-vs-tokens-definitive-guide/. Dostopano: 25. avgusta 2016.

[3] D. Hardt, B. Eaton, Y. Y. Goland, and A. Tom, »The OAuth 2.0 authorization framework,« v Internet Engineering Task Force Request For Comments, 2012. [Online]. Dosegljivo: https://tools.ietf.org/html/rfc6749. Dostopano: 14. avgust 2016.

[4] N. Sakimura, J. Bradley, M. B. Jones, B. de Medeiros, and C. Mortimore, »Final: OpenID connect core 1.0 incorporating errata set 1,« v OpenID Connect Core 1.0 Specification, 2014. [Online]. Dosegljivo: https://openid.net/specs/openid-connect-core-1_0.html. Dostopano: 14.avgust 2016.

[5] I. Sommerville, Software engineering (9th edition), 9th ed. Boston: Addison-Wesley Educational Publishers, 2010.

[6] M. Corporation, M. patterns & practices, and M. C. Staff, Microsoft application architecture for .NET: Designing applications and services. New York, NY, United States: Microsoft Press,U.S., 2009.

[7] J. Rogelj, »Upravljanje s človeškimi viri v prevajalskem procesu«, Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, Ljubljana, 2014.

[8] »IdentityServer3 - Latest version,« v IdentityServer3 Documentation. [Online]. Dosegljivo: https://identityserver.github.io/Documentation/docsv2/. Dostopano: 17. avg. 2016.

[9] A. Yaseen, »SQL server network configuration - SQL shack - articles about database auditing, server performance, data recovery, and more,« v SQL Database development,

SQL Shack - articles about database auditing, server performance, data recovery, and more, 2016. [Online]. Dosegljivo: http://www.sqlshack.com/sql-server-network-configuration/. Dostopano: 21.avgust 2016.

[10] B. Emmett, »Entity framework performance and what you can do about it - simple talk,« v SimpleTalk, Simple Talk, 2015. [Online]. Dosegljivo: https://www.simple-talk.com/dotnet/.net-tools/entity-framework-performance-and-what-you-can-do-about-it/. Dostopano: 8. avgust 2016.

[11] J. Mueller, »Understanding SOAP and REST basics and differences,« v SmartBear, 2013. [Online]. Dosegljivo: http://blog.smartbear.com/apis/understanding-soap-and-rest-basics/. Dostopano: 18. avgust 2016.

[12] »Top 10 2013-Top 10,« v OWASP, 2013. [Online]. Dosegljivo: https://www.owasp.org/index.php/Top_10_2013-Top_10. Dostopano: 24. avgusta 2016.

[13] A. van der Stock and D. Cuthbert, »OWASP Application Security Verification Standard 3.0,« v OWASP, 2016. [Online]. Dosegljivo: https://www.owasp.org/images/3/33/OWASP_Application_Security_Verification_Standard_3.0.1.pdf. Dostopano: 24.avgusta 2016.